Infineon TriCore Family On-Chip Emulation

This document is intended to be used together with the CPU reference manual provided by the silicon vendor. This document assumes knowledge of the CPU functionality and the terminology and concepts defined and explained in the CPU reference manual. Basic knowledge of winIDEA is also necessary. This document deals with specifics and advanced details and it is not meant as a basic or introductory text.

Contents

Contents **1**

1 Introduction **3**

2 Emulation options **4**
2.1 Hardware Options 4
2.2 Initialization Sequence 5
2.3 JTAG Scan Speed 5

3 CPU Setup **7**
3.1 General Options 7
3.2 Debugging Options 8
3.3 Reset 9
3.4 Aurora 10
3.5 SoC Advanced 11
3.6 SoC Options 13
3.7 Events 15
3.8 Core-specific 16

4 Internal FLASH Programming **17**

5 Debugging Multi-Core Application **17**
5.1 Multi-Core synchronization 18
5.2 Software Execution Breakpoints 18

6 Real-Time Memory Access **19**

7 Access Breakpoints **19**
7.1 TriCore V1.3 & V1.3.1 19
7.2 TriCore V1.6 23

8 Peripheral Controller Processor (PCP) **24**
8.1 winIDEA Workspaces for TriCore and PCP 24
8.2 Usage Notes 24
8.3 Reserved Resources 24

9 Trace **25**
9.1 On-Chip Trace (MCDS) Concept 25
9.1.1 Trace Memory *27*
9.1.2 Multi-Core Debug Solution (MCDS) *28*
9.2 Aurora Trace Port 31
9.3 Trace Configuration 33
9.3.1 Record everything *33*
9.3.2 Use Trigger/Qualifier *33*
9.3.3 Continuous trace recording using upload while sampling (UWS) *36*

9.3.4 Trace over HW reset *36*
9.3.5 Trace over SW reset *36*
9.3.6 TriCore Trace Qualifier Unit (TQU_TC) *37*
9.3.7 Local Memory Bus Trace Qualifier Unit (TQU_LMB) *38*
9.3.8 System Peripheral Bus Trace Qualifier Unit (TQU_SPB) *39*
9.3.9 PCP Trace Qualifier Unit (TQU_PCP) *39*
9.3.10 Central Trace Qualifier Unit (TQU_MCX) *40*
9.3.11 Time Stamp Configuration *41*

10 Profiler **44**

11 Coverage **50**

12 Getting Started **54**

13 Troubleshooting **54**

1 Introduction

Supported Tricore devices (valid September 2014):

Core	Tricore Family	Microcontroller	Emulation Device (OCT buffer size)
TC1.3		TC1766 TC1796	
TC1.3.1	AUDO FUTURE	TC1736 TC1767 TC1797	TC1736ED (256KB) TC1767ED (256KB) TC1797ED (256KB)
TC1.3.1	AUDO MAX	TC1728/24 TC1782 TC1783 TC1784	TC1782ED (512KB) TC1783ED (512KB)
TC1.6	AUDO MAX	TC1791 TC1793 TC1798	TC1798ED (768KB)
TC1.3.1	AUDO S	TC1387	TC1387ED (512KB)
TC1.6.x	AURIX	TC23xLP TC26xD TC274D TC27xT TC27xTP TC29xT TC29xTP TC2D5T	TC26xDE (512KB) TC27xTE (1MB) TC27xTF (1MB) TC29xTE (1MB) TC29xTF (1MB) TC2D5TED (512KB)

Contact iSYSTEM for the latest list of supported devices or check Supported MCUs section at www.isystem.com.

The development tool connects to the on-chip OCDS (On-Chip Debug Support) via 4 or 5-wire JTAG, or via 2 or 3-wire DAP debug interface. Not all variants of debug interface are available on every microcontroller.

A standard JTAG interface according to IEEE1149.1 is provided for chip and board testing as well as for debugging and calibration using legacy equipment.

The cost inferred by each non-functional pin is a strong argument to reduce the tool access port to as few pins as possible. The standardized device access port (DAP) of all current Infineon microcontrollers offers a convenient method to get the required functionality at the least possible cost. Optional 3-wire DAP interface (Wide Mode) is supported on certain devices for improved performance.

In practice, the DAP interface proves to be faster since it can operate at higher clock and e.g. new AURIX devices additionally support optional 3-wire DAP interface.

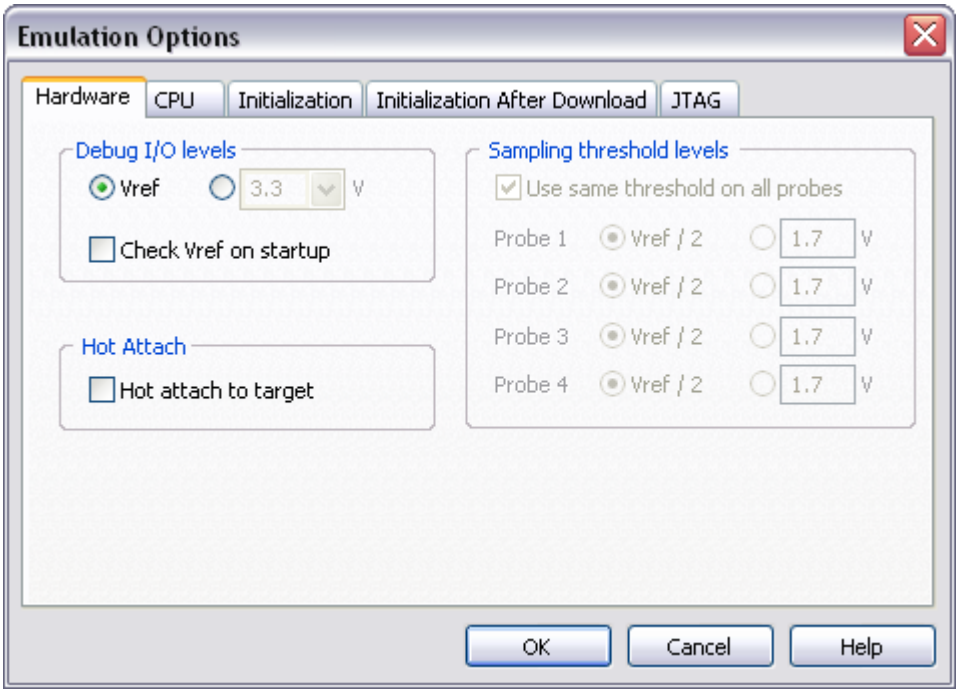
Due to the lack of the trace functionality on standard Tricore devices, Infineon provides a dedicated pin compatible Emulation Devices, where trace, based on up to 1MB on-chip trace buffer is available. This allows using the Emulation Device on a regular target during the development process when the trace functionality is required.

Debug Features

- DAP and JTAG debug interface
- 4 on-chip hardware execution breakpoints (TriCore V1.3 & V1.3.1)
- 8 on-chip hardware execution breakpoints (TriCore V1.6 & V1.6.x)
- Unlimited software breakpoints including in the flash
- Access breakpoints
- Real-time access
- Internal/external flash programming
- Trace, Profiler and Execution Coverage on Emulation Devices (ED)

2 Emulation options

2.1 Hardware Options



Emulation options, Hardware pane

Debug I/O levels

iC3000 development system can be configured in a way that JTAG or DAP debug signals are driven at 3.3V, 5V or target voltage level (Vref). In case of ic5000 development system, arbitrary voltage can be set beside the listed 2.5, 3.3V and 5V by simply writing custom value e.g. 2.8V in the same field.

When 'Vref' Debug I/O level is selected, a voltage applied to the belonging reference voltage pin on the target debug connector is used as a reference voltage for voltage follower, which powers buffers, driving the debug
JTAG/DAP signals. The user must ensure that the target power supply is connected to the Vref pin on the target JTAG/DAP connector and that it is switched on before the debug session is started. If these two conditions are not meet, it is highly probably that the initial

debug connection will fail already. However in some cases it may succeed but then the system will behave abnormal.

It is recommended to use ‘Vref’ setting for target Vref voltages 3.3V and below. When debug I/O levels should be 5V, it is recommended selecting internal ‘5.0V’ source for Debug I/O levels.

Check Vref on startup (iC5000 only)

iC5000 development system can measure voltage on the Vref pin on the target debug connector. When 'Vref' Debug I/O level is selected, the debugger will pop up a warning message if no voltage is detected on the target debug connector.

Hot Attach

Option must be checked when Hot Attach is used.

2.2 Initialization Sequence

In case of Aurix family, initialization sequence usage is obligatory since the target microcontroller needs to be preconfigured for operation in conjunction with the debugger. Suspend control of used peripheral modules, MCDS (on-chip trace logic) and multi-core synchronization must be configured via the initialization sequence. Initialization files are provided for specific Aurix devices in winIDEA Examples directory. Examples may be installed from winIDEA Help menu (manually on demand). Search for path:

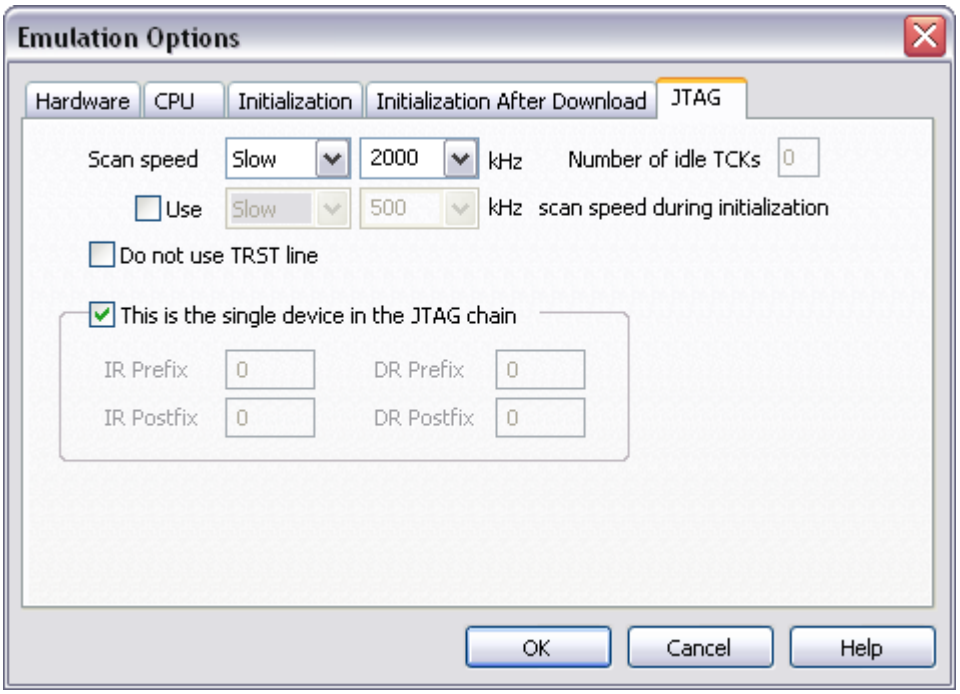
examples\OnChip\TriCore\CPU\AURIX\Obligatory_AURIX_Initilization_Files

in the examples installation folder, find README.txt and read the detailed explanation.

Detailed information on initialization sequence configuration may be found in the Initialization Sequence help topic.

Note: Keep the default ‘Specify’ and ‘0’ setting for Address offset within the Initialization tab.

2.3 JTAG Scan Speed



JTAG Scan Speed definition

Note: These settings are not available when DAP debug interface is used.

Scan speed

The JTAG chain scanning speed can be set to:

- Slow - long delays are introduced in the JTAG scanning to support the slowest devices. JTAG clock frequency varying from 1 kHz to 2000 kHz can be set.
- Fast – the JTAG chain is scanned with no delays.
- Other scan speed types (not supported for Tricore family) can be seen and are automatically forced to Slow.

Slow and Fast JTAG scanning is implemented by means of software toggling the necessary JTAG signals.

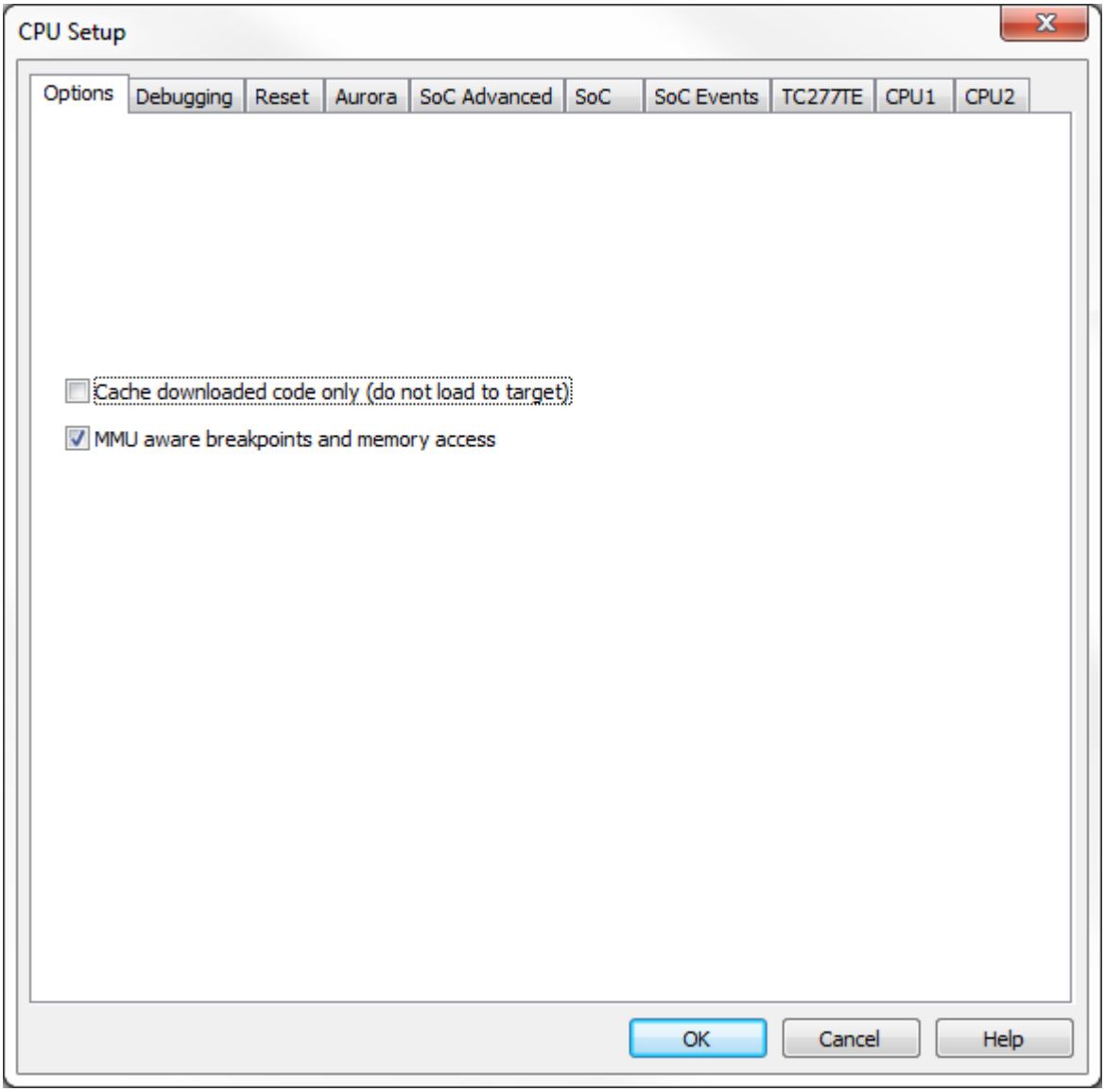
In general, Fast mode should be used as a default setting. If Fast mode fails or the debugging is unstable, try Slow mode at different scan frequencies until you find a working setting.

Use – Scan Speed during Initialization

On some systems, slower scan speed must be used during initialization, during which the CPU clock is raised (PLL engaged) and then higher scan speeds can be used in operation. In such case, this option and the appropriate scan speed must be selected.

3 CPU Setup

3.1 General Options



Cache downloaded code only (do not load to target)

When this option is checked, the download files will not propagate to the target using standard debug download but the Target download files will.

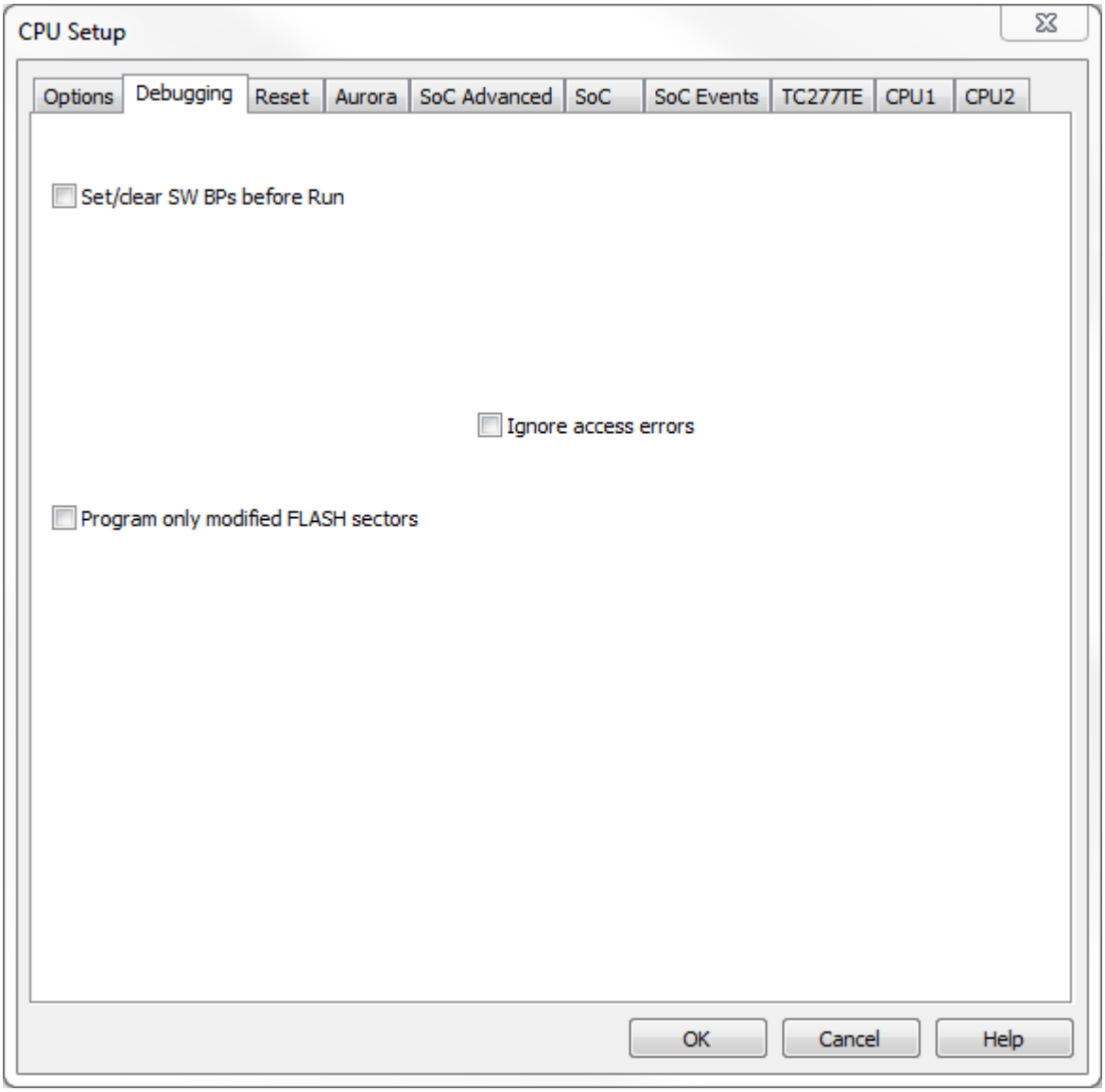
In cases, where the application is previously programmed in the target or it's programmed through the flash programming dialog, the user may uncheck 'Load code' in the 'Properties' dialog when specifying the debug download file(s). By doing so, the debugger loads only the necessary debug information for high level debugging while it doesn't load any code. However, debug functionalities like ETM and Nexus trace will not work then since an exact code image of the executed code is required as a prerequisite for the correct trace program flow reconstruction. This applies also for the call stack on some CPU platforms. In such applications, 'Load code' option should remain checked and 'Cache downloaded code only (do not load to target)' option checked instead. This will yield in debug information and code image loaded to the debugger but no memory writes will propagate to the target, which otherwise normally load the code to the target.

MMU aware breakpoints and memory access

If checked, breakpoints are set with physical addresses, memory accesses are performed using virtual/physical mapping.

If the option is cleared, no distinction is made between physical and virtual addresses.

3.2 Debugging Options



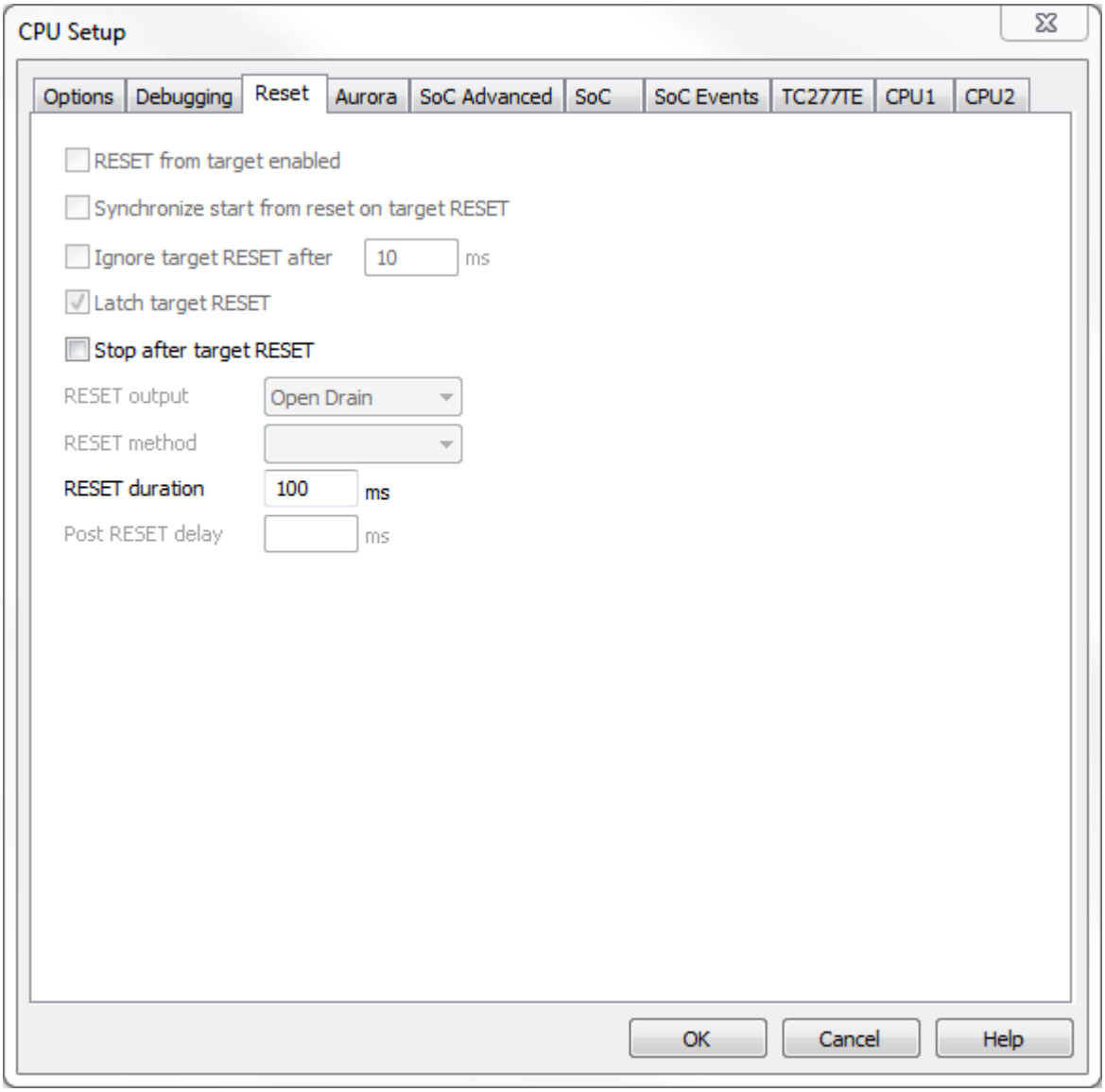
Set/clear SW BPs before Run

When the option is checked, then a software breakpoint is not set/cleared immediately, but is just remembered. Only when the CPU is set to running are the breakpoints committed. This way several breakpoints can be changed but only one re-FLASH operation takes place. This is especially noticeable in testIDEA operation with many stubs and also during a regular debugging session when several breakpoints are set/cleared within the same flash erase block.

Program only modified FLASH sectors

Optionally, when a download is performed into an already programmed device, only the FLASH sectors which have been modified are programmed. This speeds up download times when only small changes are performed.

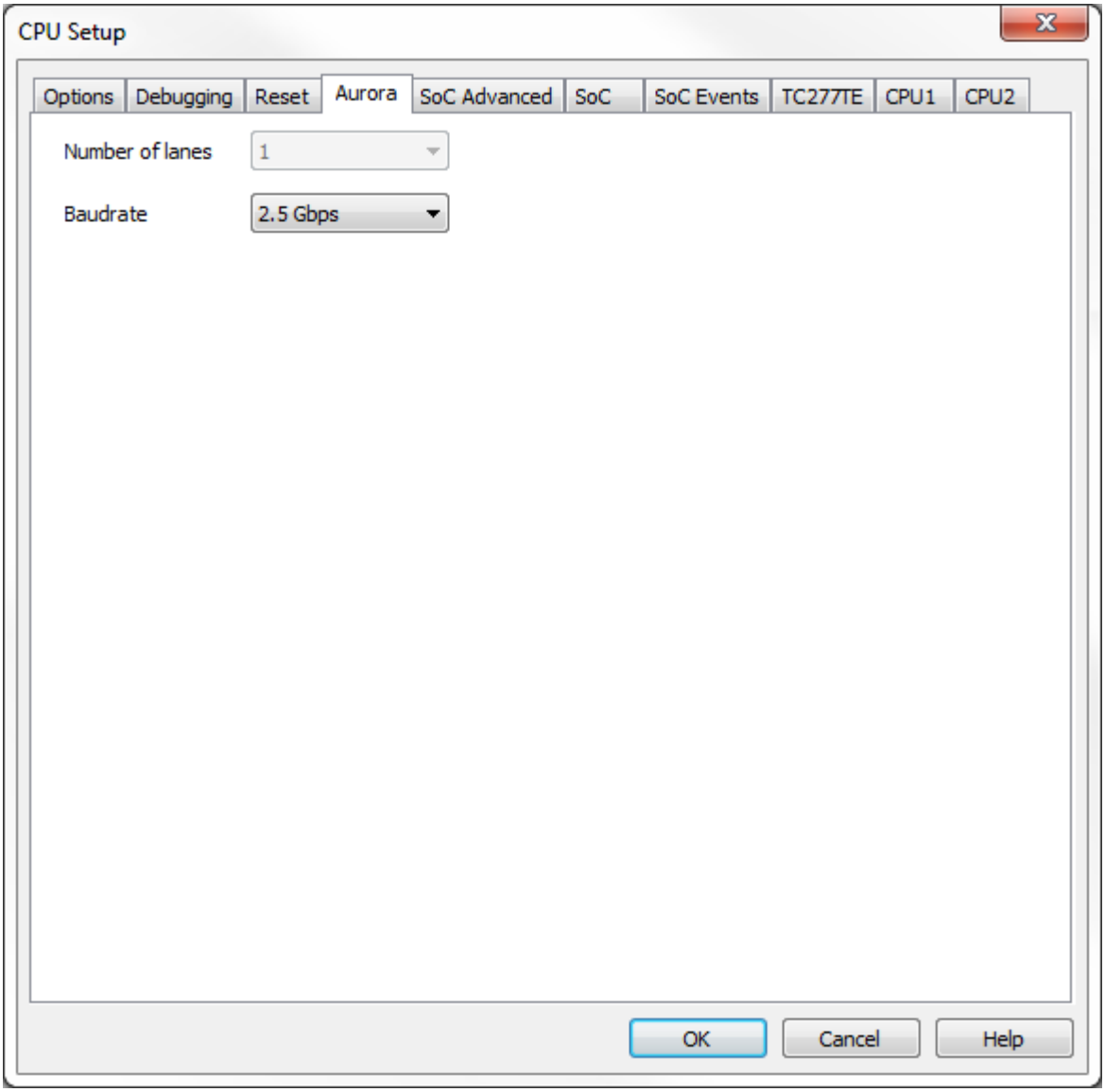
3.3 Reset



RESET Duration

The width of the RESET pulse is specified here.

3.4 Aurora



Number of lanes

Select the maximum number available on your target configuration. This depends on the TriCore device used, but may be additionally limited if only a portion of lanes is physically connected to the Aurora trace port connector on the target board.

Note: First TriCore emulation devices (TC27xTE, TC29xTE) implement 1 lane Aurora interface. For these devices the selection is disabled and forced to 1.

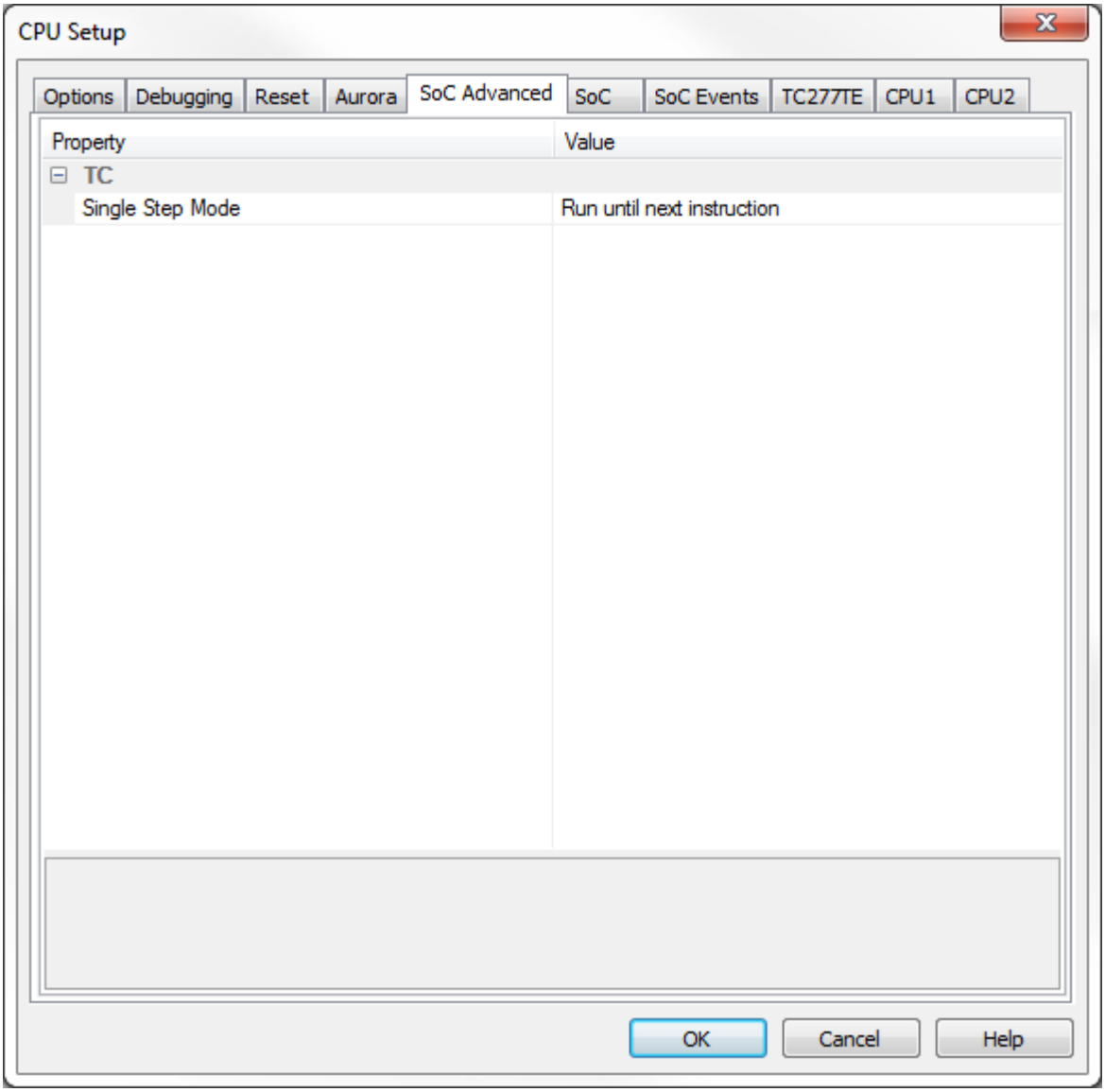
Baudrate

Select a maximum baudrate available. Aurora works in Gbit/s range, so the target board needs to be carefully designed and manufactured. If any issues arise, lower the baudrate to the range that is acceptable for your target board design.

3.5 SoC Advanced

Single Step Mode

On-chip debug logic does not directly implement single instruction step. Hence the debugger must implement instruction step by analyzing each instruction and calculating all possible program addresses where program continues or jumps to. Note that conditional instruction can take multiple program flow paths depending on the condition at run time. When executing instruction step, execution breakpoints are set on the calculated program addresses, then the CPU is run which stops just after one instruction on one of the pre-set execution breakpoints. All these execution breakpoints management and run/stop control are performed hidden from the user. With this concept, interrupts can't be disabled during the instruction step. This means that any interrupt, which is pending at the time of the instruction step, will be serviced immediately. Best debugging experience is achieved by **suspending the microcontroller peripherals** while the CPU is stopped. This can be done by an initialization script, such as the one shown in the Multi-Core synchronization chapter. This way interrupts are not generated while the CPU is stopped and the debugging is close to the real-time operation as much as possible.



If the peripherals are not suspended when the CPU is stopped, then the interrupts may kick when the instruction step is being performed. The debugger offers two different methods of handling such situations:

- **Run until next instruction**

Breakpoint is set on the next instruction as explained earlier and the CPU is set to run. This is the default operation mode. Pending interrupts will be serviced before the CPU stops on the breakpoint.

This method will not work, if the interrupt service routine of the pending interrupt is not written properly since then it can happen that the program does not return from the interrupt routine back to the original program flow and none of the calculated and pre-set execution breakpoints (part of the single step algorithm) will ever hit. User would see this 'instruction step' operation like program continuously running instead of stopping after one instruction being executed. To troubleshoot such a case, an alternative 2nd method is available.

- **Stop after op-code executed**

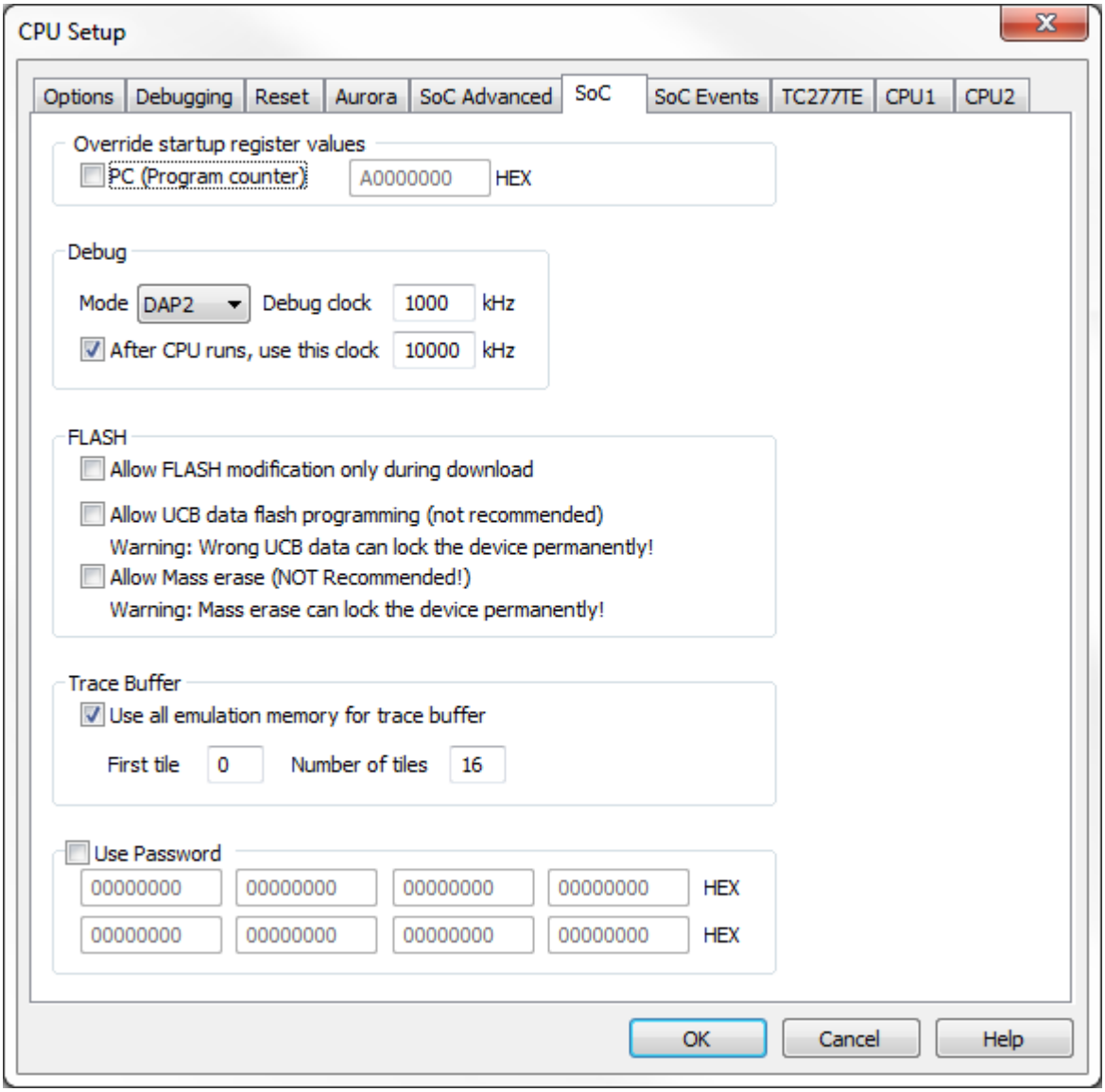
Execution breakpoint logic is configured in a way that the CPU stops immediately after the first instruction is being executed.

If no interrupts are active respectively all belonging peripherals are suspended, a single instruction in question will be executed.

However, if there is a pending interrupt at the moment when the user executes instruction step, the program will typically stop on the first instruction within the interrupt routine. This makes instruction stepping practically impossible, since interrupts could be pending all the time. For this reason this method is not suitable for general use and should be used only when experiencing problems with the first method.

A use case was described under the first method, where pending interrupt routine does not return properly and the program goes into 'running'. By selecting the 2nd method, the problematic interrupt routine will be pinpointed since the program will stop at the beginning of the problematic interrupt routine from which it otherwise doesn't seem to return as anticipated.

3.6 SoC Options



Override startup register values

This option overrides the default Instruction Pointer reset value with the value set.

Debug

Tricore device can feature JTAG, DAP and DAP2 debug interfaces. JTAG, DAP or DAP2 debug mode must be selected depending on the debug interface used. Additionally, DAP clock must be selected when DAP or DAP2 debug interface is used. Typical DAP clock is in range between 2 to 4MHz.

The debugger must be connected to the target DAP connector (10-pin 1.27 mm pitch) when DAP or DAP2 debug mode is selected and to the target JTAG connector (16-pin 2.54mm pitch) when JTAG debug mode is selected.

Note: DAP2 debug interface is supported on iC6000 and iC5000 in conjunction with the IC50163 (DAP2 Wide Cable Adapter). It not supported in conjunction with older IC50161 (DAP Cable Adapter).

After CPU runs, use this clock

Frequency of the DAP debug interface is limited by the microcontroller system clock. Since the microcontroller starts with relatively low frequency after exiting the reset state, the initial DAP clock frequency must be low too. Once the application engages the PLL, the system clock typically goes to 90MHz and above, which means that also the DAP debug interface can run at higher frequency, yielding better throughput when uploading the MCDS (on-chip trace) buffer to the PC. Typically, this option is used in conjunction with the MCDS configure for “upload while sampling”.

Note: This option is available for DAP and DAP2 debug interface only.

Allow flash modification only during download

When this option is checked, internal flash can be modified only through the debug download. When unchecked, it can be modified via memory window too.

Allow UCB data flash programming (not recommended)

When this option is checked, the UCB (User Configuration Block) can be modified through the memory window or through the target download. Note that wrong UCB data can lock the device permanently. Therefore use it at your own risk.

Allow Mass erase (not recommended)

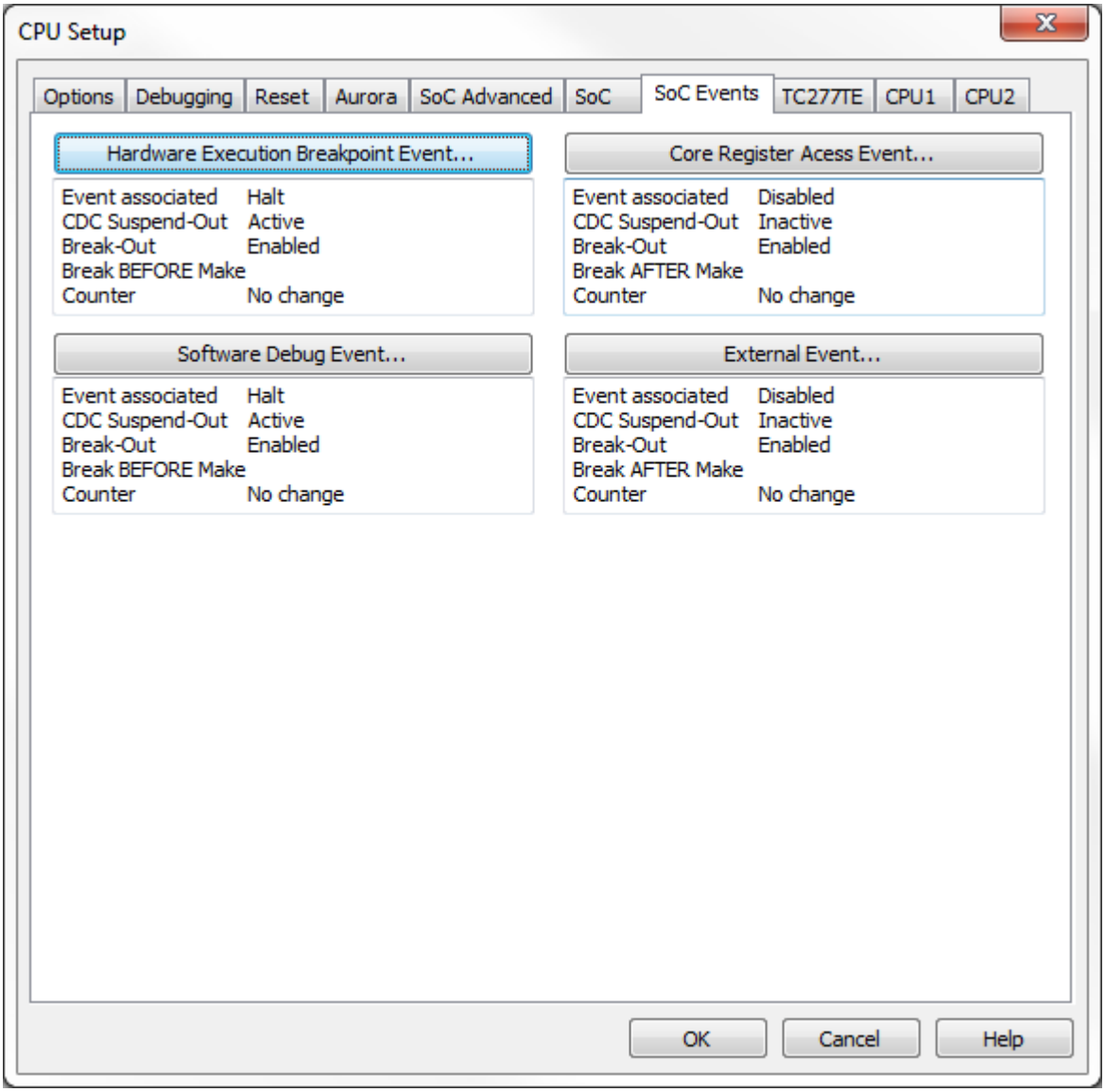
When this option is checked, Mass erase may be performed. Note that mass erase can lock the device permanently; therefore use it at your own risk.

Trace Buffer

By default ‘Use all emulation memory for trace buffer’ option is checked. In this case, all emulation memory (EMEM) is used for trace. Alternatively this memory can be also used for calibration via overlay of a non-volatile memory. When the emulation memory is split between the calibration and the trace, user must define, which tiles of the emulation memory are available for the trace. Note that the trace requires a block of consecutive tiles.

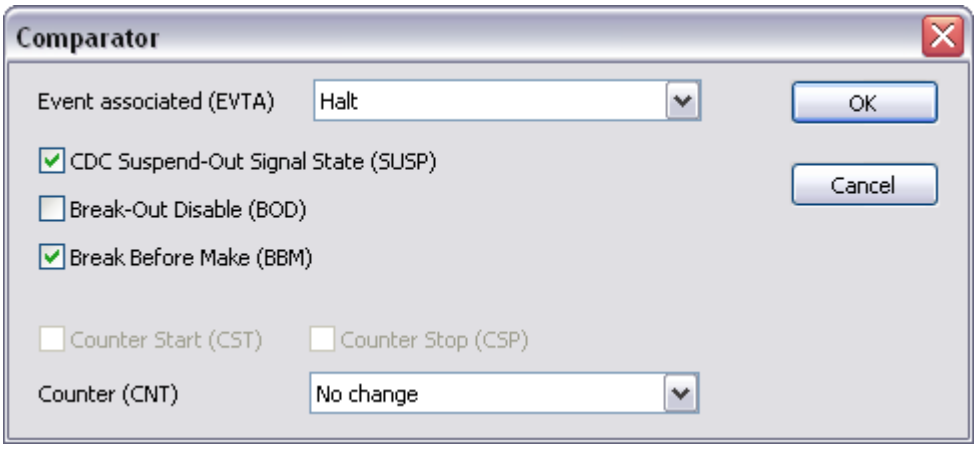
Note: Be cautious when Emulation Memory is being used by the application. If the target application would inadvertently use the same tiles of Emulation Memory, which were allocated for tracing, trace could behave unexpectedly or would not work at all.

3.7 Events



It is possible to configure individual event actions for hardware execution breakpoint event (TR0EVT, TR1EVT), core register access event (CREVT), software debug event (SWEVT) and external break input event (EXEVT).

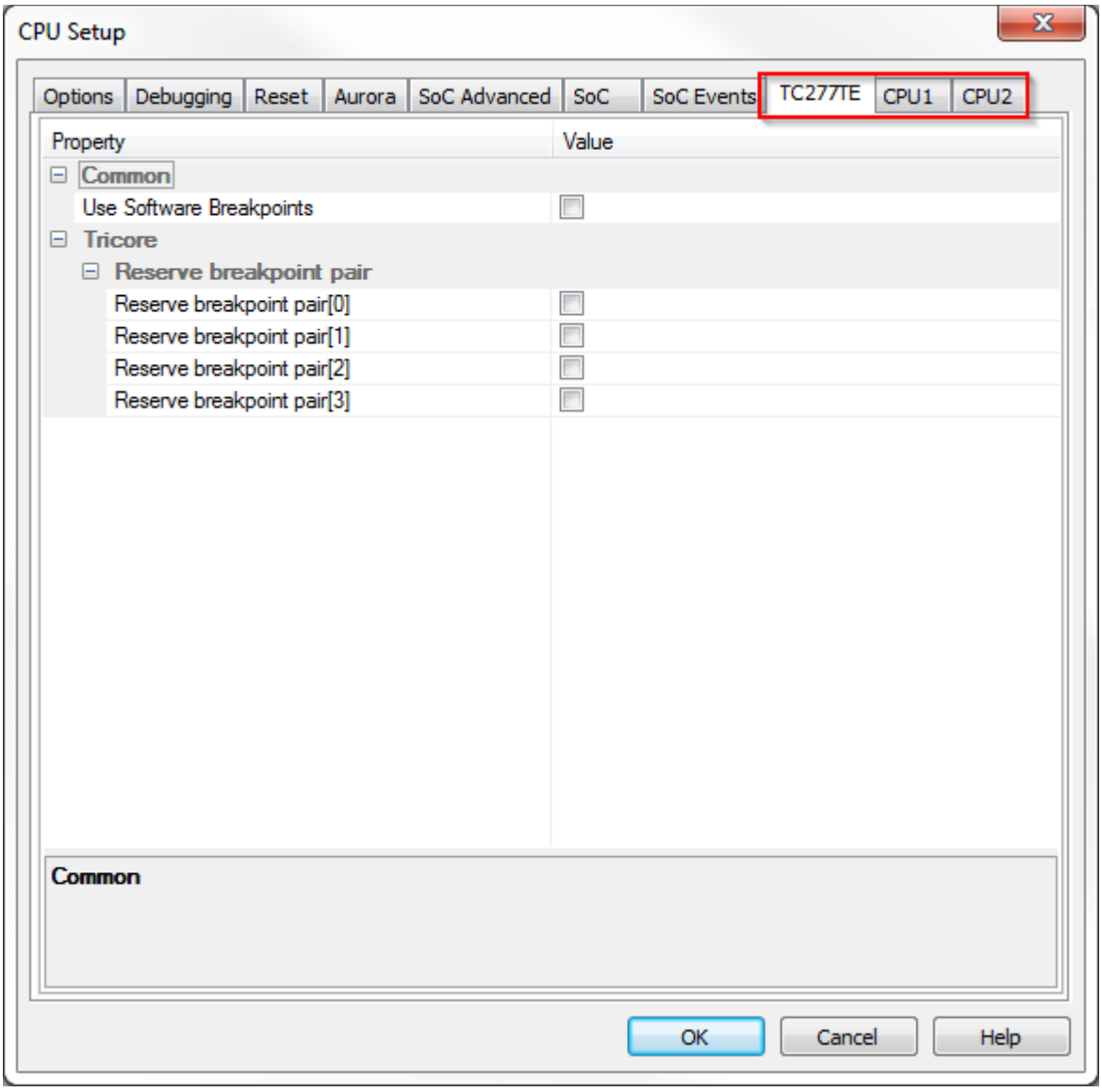
Press the button to open the Event dialog.



Refer to Access Breakpoints chapter for available event actions and explanation.

3.8 Core-specific

Equivalent dialog is provided for each available CPU core:



Execution Breakpoints

Hardware Breakpoints

Hardware breakpoints are breakpoints that are already provided by the CPU. The number of hardware breakpoints is limited to four. The advantage is that they function anywhere in the CPU space, which is not the case for software breakpoints, which normally cannot be used in the FLASH memory, non-writeable memory (ROM) or self-modifying code. If the option 'Use hardware breakpoints' is selected, only hardware breakpoints are used for execution breakpoints.

Note that the debugger, when executing source step debug command, uses one breakpoint. Hence, when all available hardware breakpoints are used as execution breakpoints, the debugger may fail to execute debug step. The debugger offers 'Reserve one breakpoint for high-level debugging' option in the Debug/Debug Options/Debugging' tab to circumvent this. By default this option is checked and the user can uncheck it anytime.

Note: Memory protection unit must not be used while debugging the application. Hardware breakpoints and instruction step cannot be used in conjunction with Memory Protection Unit.

Software Breakpoints

Available hardware breakpoints often prove to be insufficient. Then the debugger can use unlimited software breakpoints to work around this limitation.

When a software breakpoint is being used, the program first attempts to modify the source code by placing a break instruction into the code. If setting software breakpoint fails, a hardware breakpoint is used instead.

The debugger provides software breakpoints in the internal program flash too but they are almost useless since only erasing a single erase unit can take up to 5s already.

Reserve breakpoint pair

One of possible operation modes for each of these events is execution breakpoint mode and by default all 8 eight are used by the debugger. When any (configurable in pairs) of these is not allowed to be used by the debugger, they can be reserved through these settings.

4 Internal FLASH Programming

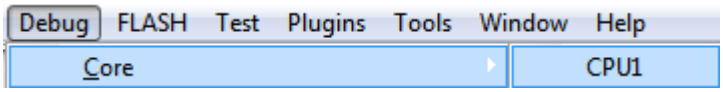
Internal CPU program flash is programmed through the standard debug download. The debugger recognizes which code from the download file fits in the internal flash and programs it during the debug download.

A standard FLASH setup dialog accessible from the FLASH menu is used only for programming external flash devices.

5 Debugging Multi-Core Application

When a microcontroller features two or more cores, each core is debugged in an individual winIDEA instance. In order to debug the non-primary core (for example CPU1 on TC277T), first primary winIDEA instance and workspace is opened, which allows compiling the complete project, downloading the code in the MCU program flash and debugging the primary (main) core.

In order to debug the 2nd core (CPU1 in the below screenshot), new winIDEA instance is opened from the **Debug** / **Core** menu.



2nd core is debugged more or less in the same way as a primary core. 2nd core winIDEA instance provides all standard debug windows such as disassembly window, memory window, watch window, variable window, source window, trace window, etc.

The application code for 2nd (and any further) core is loaded by the primary winIDEA instance/workspace, which downloads the application code in the MCU internal program flash. Program flash is shared amongst all cores available. 2nd (and any further) core winIDEA instance requires to download only symbols for the specific core being debugged. Don't forget to specify the necessary download file including debug symbols in each non-primary core winIDEA instance.

When the microcontroller is released from the reset state, all cores are stopped at start point. All non-primary cores are started by the primary core application code.

5.1 Multi-Core synchronization

Some Aurix microcontrollers have multiple cores, which are not synchronized automatically after reset. Cerberus (CBS) module may be used to synchronize the cores, for example to stop all cores when an execution breakpoint in one core is hit. As specified in the Aurix User's Manual, all CPUs should become sensitive to suspend signal and all CPUs should provide suspend triggers. This can be done by using a winIDEA initialization script (**Hardware** / **Emulation Options** / **Initialization**). Below is an example initialization sequence for the MCU with three cores:

```
// MULTI CORE SYNCHRONIZATION
A CBS_TLCHE L  0x00000002 //TL1 capture and hold enabled
A CBS_TL1ST L  0x30000007 //all CPUs are suspend target
A CBS_TRC0  L  0x00000102 //BT1 - CPU0 is trigger source,
                          //HALT connected to TL2
A CBS_TRC1  L  0x00000100 //BT1 - CPU1 is trigger source
A CBS_TRC2  L  0x00000100 //BT1 - CPU2 is trigger source
A CBS_TLC   L  0x00000030 //TL1 forced to active
A CBS_TLC   L  0x00000000 //TL1 force removed
```

Additionally, the peripheral modules can be suspended along with the cores too. In order to be able to suspend the peripheral modules on a suspend signal, some need to be enabled before they can be configured. Note that certain peripheral modules are enabled out of reset but not all. The following initialization script suspends the three STM timers, which are typically used for periodic interrupts.

```
//STM suspend control – already enabled after reset
A STM0_OCS      L  0x12000000 //Module is suspend target
A STM1_OCS      L  0x12000000 //Module is suspend target
A STM2_OCS      L  0x12000000 //Module is suspend target
```

All these help for better user experience when using the debugger.
To summarize, for a predictable application behavior (from user point of view) in conjunction with the connected debugger, certain microcontroller logic and modules have to be preconfigured before the application can be debugged. See 2.2 Initialization Sequence chapter for more details on obligatory Aurix pre-configuration.

5.2 Software Execution Breakpoints

Software execution breakpoints are not available on non-primary cores since setting and clearing a software breakpoint means reprogramming (erase-program) the program flash.

Any non-primary core could in the meantime, while software execution breakpoint is being applied, try to execute the code from the same flash sector, which would result in the application going into weeds.

Debugger still provides software execution breakpoints for the primary (main) core. They can be used when debugging a single core application (either on a single or a multi-core microcontroller). However, as soon as the application runs on multiple cores, software execution breakpoints should not be used to prevent problems mentioned in the previous paragraph.

6 Real-Time Memory Access

TriCore debug module supports real-time memory access. Watch window's Rt.Watch panes can be configured to inspect memory with minimum intrusion while the application is running. Optionally, memory and SFR windows can be configured to use real-time access as well.

In general it is not recommended to use real-time access for Special Function Registers (SFRs) window. In reality, real-time access still means stealing some CPU cycles. As long as the number of real-time access requests stays low, this is negligible and doesn't affect the application. However, if you update all SFRs or memory window via real-time access, you may notice different application behavior due to stealing too many CPU cycles.

When a particular special function register needs to be updated in real-time, put it in the real-time watch window (don't forget to enable real-time access in the SFRs window but keep SFRs window closed or open but with SFRs collapsed). This allows observing a special function register in real-time with minimum intrusion on the application.

Using "alternative" monitor access to update a memory location or a memory mapped special function register while the application is running works like this: the application is stopped, the memory is read and then the application is resumed. Hence the impact on real time execution is severe and use monitor access for 'update while running' only if you are aware of the consequences and can work with them.

7 Access Breakpoints

7.1 TriCore V1.3 & V1.3.1

Note: Memory protection unit must not be used while debugging the application on TriCore V1.3 and V1.3.1 based devices. If the Memory Protection Unit is being used, the application being debugged will behave unpredictably since hardware execution and access breakpoints are implemented by using Memory Protection Unit (it is by design). These restrictions don't apply for TriCore V1.6 core.

The CDC (Core Debug Controller) allows code and data triggers to be combined to create a Debug Event. The combination is specified by the Trigger Event Register (TR0EVT and TR1EVT). The Trigger Event Unit can generate a number of Trigger Debug Events by combining four Debug Triggers for each Trigger Debug Event. The Debug Triggers are generated by the memory protection system. There are four possible Debug Triggers generated by the Memory Protection System:

DLR

Data read or write access to the lower bound or range of the Data RTEn,as enabled in the Data Protection Mode (DPM) register.

DU

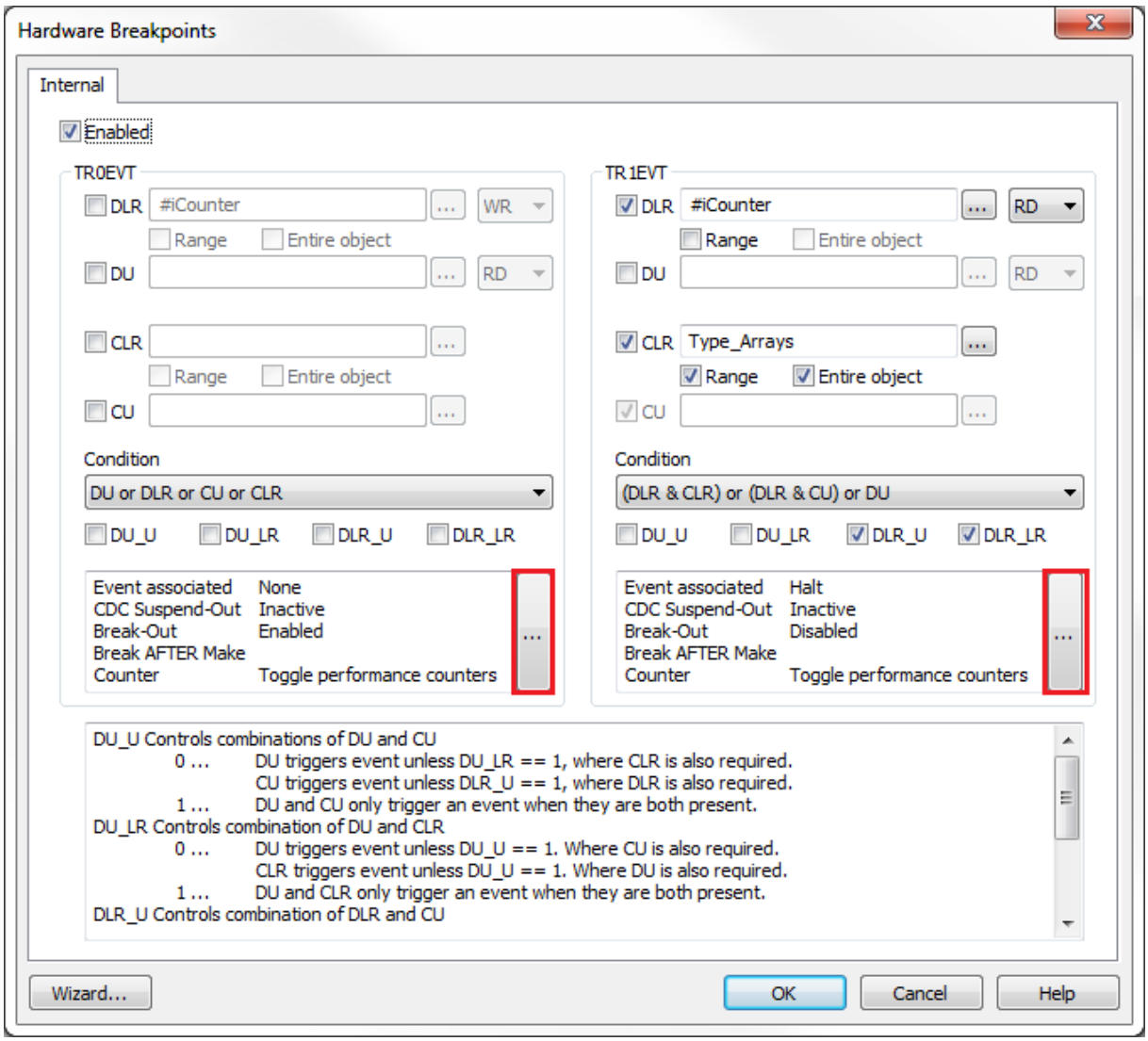
Data read or write access to the upper bound of the Data RTEn, as enabled in the Data Protection Mode (DPM) register.

CLR

Code execution from the lower bound address or address range of the Code RTEn, as enabled in the Code Protection Mode (CPM) register.

CU

Code execution from the upper bound address of the Code RTEn, as enabled in the Code Protection Mode (CPM) register.



Tricore Access Breakpoints

Condition

Selects debug trigger combination which generates a Debug Event. 16 different combinations are possible.

DU_U, DU_LR, DLR_U, DLR_LR

DU_U controls combinations of DU and CU, DU_LR controls combination of DU and CLR., DLR_U controls combination of DLR and CU and DLR_LR controls combination of DLR and CLR.

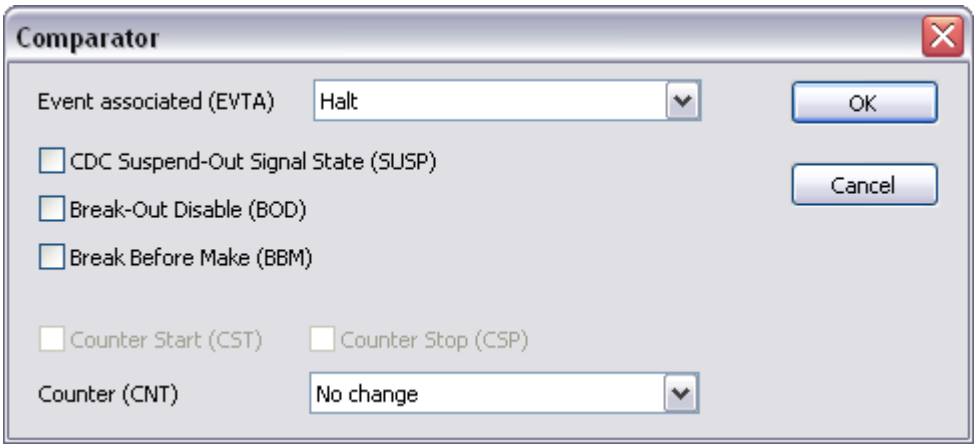
In principle these four options are unnecessary because matching setting of the debug trigger combination can be selected in the Condition combo box no matter how these options

are set. They are available just not to restrict the user from programming these bit fields in the Trigger Event registers TR0EVT and TR1EVT.

Refer to the Target Specification document of specific TriCore microcontroller for more details on using these options.

Event Action

Different actions can be configured for individual Trigger Event register. Default configuration stops the core only. Event dialog is open by pressing the button in the right bottom corner of the TR0EVT and TR1EVT configuration area (encircled in red in the Tricore Access Breakpoints dialog in the previous page).



Event Associated (EVTA)

Disabled

The event is disabled and no actions occur: the suspend-out signal and performance counter control ignore the event.

None

No action is implemented through the EVTA field of the event's register however the suspend-out signal and performance count still occur as normal for an event.

Halt

The Debug Action Halt, causes the Halt mode to be entered where no more instructions are fetched or executed. While halted, the CPU does not respond to any interrupts.

Breakpoint Trap

The Breakpoint Trap enters a Debug Monitor without using any user resource. Refer to TriCore Core Architecture manual for more details.

Breakpoint Interrupt

One of the possible Debug Actions to be taken on a Debug Event, is to raise a Breakpoint Interrupt. The interrupt priority is programmable and is defined in the control register associated with the breakpoint interrupt. Refer to TriCore Core Architecture manual for more details.

CDC Suspend-Out Signal State (SUSP)

The suspend-out signal is asserted when a debug event occurs. It is up to the user then to configure according peripheral module to act upon asserted suspend-out signal.

Break-Out Disable (BOD)

When this option is checked, BRKOUT signal is not asserted. This takes priority over any assertion generated by the EVTA field.

Break Before Make (BBM)

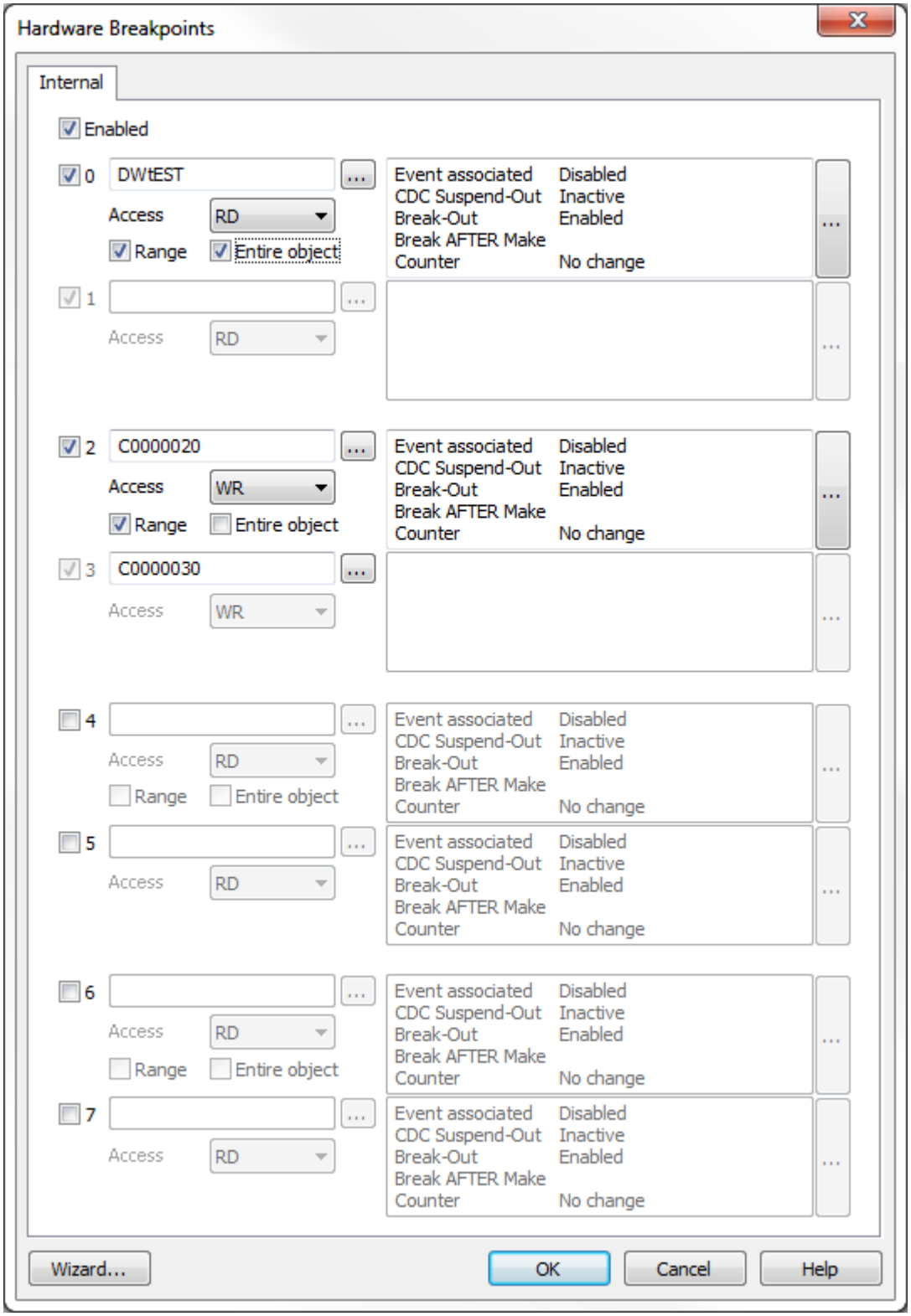
When this option is checked, Halt mode performs a cancel of all instructions after and including the instruction that caused the breakpoint. If unchecked, it cancels all instructions after the instruction that caused the breakpoint.

Counter (CNT)

When the performance counter is operating in task mode, the counters are started and stopped by debug actions. All event registers allow the counters to either be started or stopped.

The trigger event registers also allow the mode to be toggled to active (start) or inactive (stop). This allows a single Range Table Entry (RTE) to be used to control the performance counter in certain applications.

7.2 TriCore V1.6



It's possible to configure access breakpoints on up to 8 individual addresses or up to 4 address ranges. Access type (RD, WR, RD/WR, Instruction) and Event Action can be selected for each access breakpoint. While possible access types are self-explanatory, read Event Action explanation in the previous chapter since it's valid for TriCore V1.6 too.

8 Peripheral Controller Processor (PCP)

This chapter discusses debugging of the Infineon Peripheral Control Processor, PCP for short, within the iSYSTEM winIDEA environment. PCP features and debugging methods are device implementation dependent. Supported TriCore devices at the time of this writing (January 2010) are: TC1767, TC1767ED, TC1797 and TC1797ED.

8.1 winIDEA Workspaces for TriCore and PCP

Debugging session is started by opening a TriCore winIDEA workspace, e.g. Sample.xjrf. The download file contains both a TriCore run-time image as well as a PCP run-time code image. Set a breakpoint in the **`main.c`** module after the initialization of the PCP and other related special function registers, timers and I/O ports, for example.

At this point also the PCP run-time image has been copied into the PCP CMEM code memory by the compiler startup code. Now click on Debug/Core/PCP. This will open a new, secondary instance of winIDEA. It will automatically load a PCP workspace. The name of this workspace is defined by the TriCore workspace name of the primary winIDEA, to which a _PCP extension is appended. In our case this results to Sample_PCP.xjrf.

On Debug/Download winIDEA will load symbols for the PCP debug. It will also display a register context for the channel number 1 by default and disassemble a PCP code, if the PCP has been enabled in the PCP_CS.EN control/status register bit.

8.2 Usage Notes

Note that PCP instructions can disable debugging a PCP channel by clearing its saved context R7 register, the channel enable bit R7.CEN. A warning is displayed if run or step is attempted. To continue debugging, the bit needs to be re-set by hand.

The PCP debugging provides only software breakpoints. There are no hardware breakpoints.

Whenever TriCore CPU is reset and run again, the breakpoints set in the PCP winIDEA session will be overwritten. Open the Breakpoints dialog and click on the Reapply All.

Use F8 (Debug/Snapshot) to refresh windows' contents in the other winIDEA instance when monitoring shared TriCore and PCP memory resources.

Any access to an undefined memory space, for example access beyond the implemented CMEM or PRAM space, causes a bus error in TriCore. To recover, the CPU has to be reset.

When downloading code that has already been programmed into flash, the *Cache downloaded code only* option in the CPU Setup dialog can be used to bypass redundant and slow flash programming. Do, however, remember to uncheck this option whenever the project is recompiled.

In certain cases winIDEA will display the PCP status STOP, when in fact the PCP is running. This happens when a PCP channel program is interrupt-driven from the TriCore. For example, in the BlinkLED sample the interrupt is issued in the order of 500ms, while the PCP program takes only a couple of 100ns to execute. When the PCP execution stops, it waits for another invocation by the TriCore interrupt routine. Therefore, most of the time the PCP is in waiting, and this is perceived by a debugger as being stopped.

8.3 Reserved Resources

For the PCP run and step control winIDEA is using the CPU_SBSRC, the CPU Software Breakpoint Service Request Control Register, at address F7E0.FFBCh. User application should not use this register.

The current PCP channel being debugged is selected with the SRPN bit-field of the CPU_SBSRC register. When needed, this scope can be changed either in the Disassembly Registers Window, where the channel service request field is marked with a CH, or directly in the SFR window where the register is located in the CDR Core Debug Register Group.

9 Trace

Refer to winIDEA Contents Help, Analyzer Window section (or alternatively to the standalone Analyzer.pdf document) for general information on Trace user interface and use.

Per default, TriCore target processors don't provide debug trace functionality, which is often necessary during the development and test process. As an alternative, Infineon offers a dedicated pin compatible Emulation Device (ED), which features Nexus compliant on-chip trace (MCDS) in conjunction with the standard OCDS debug module, which is controlled by the external tool through the JTAG or DAP debug interface.

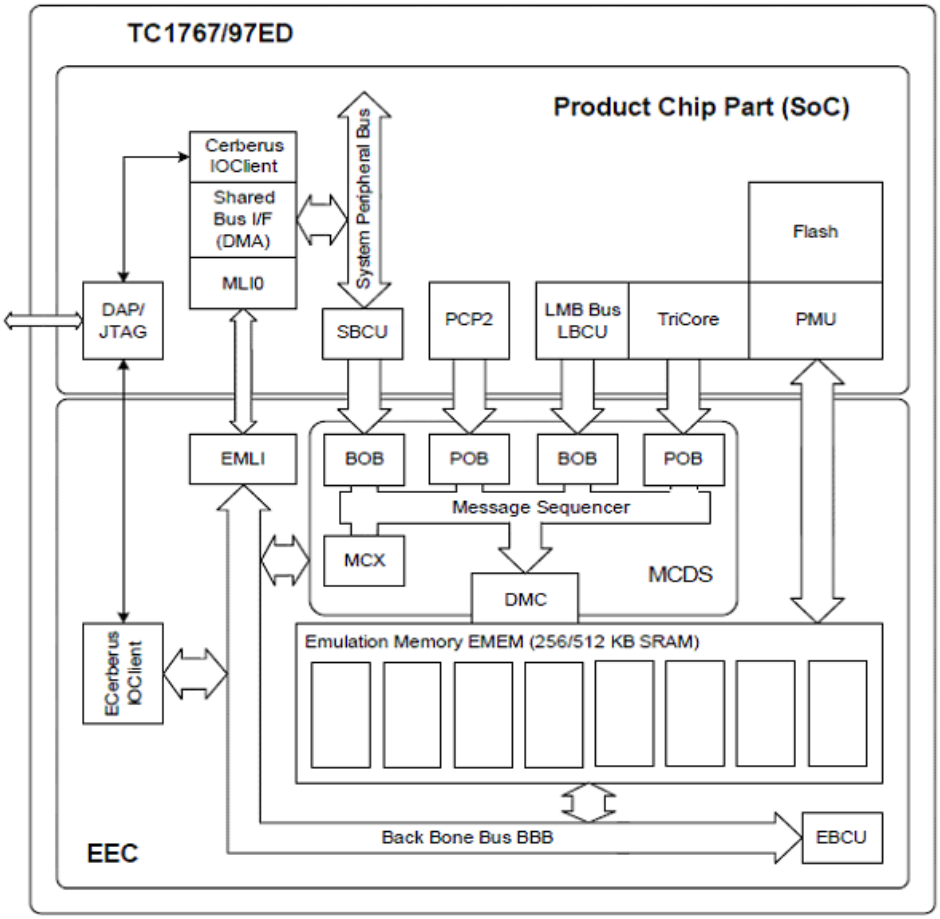
Emulation device can feature either:

- Aurora trace port, where Nexus trace information is pushed immediately off-chip to the external debug tool
- On-Chip Trace Buffer, where Nexus trace information is stored in a dedicated trace buffer until it's full and then read through debug interface and uploaded to the PC for further analysis.

Devices may support one or both of these two trace technologies. Availability depends on the specific TriCore (ED) emulation device.

9.1 On-Chip Trace (MCDS) Concept

Some technical background on the MCDS is fundamental in order to use highly capable but relatively complex TriCore trace. Majority of the text explaining the MCDS is taken from the Infineon TC1767ED documentation and its sole purpose is to get the user acquainted with the MCDS, which will then ease the trace use supported by iSYSTEM development tool. Contact Infineon representative when a detailed knowledge on the Emulation Device is required.



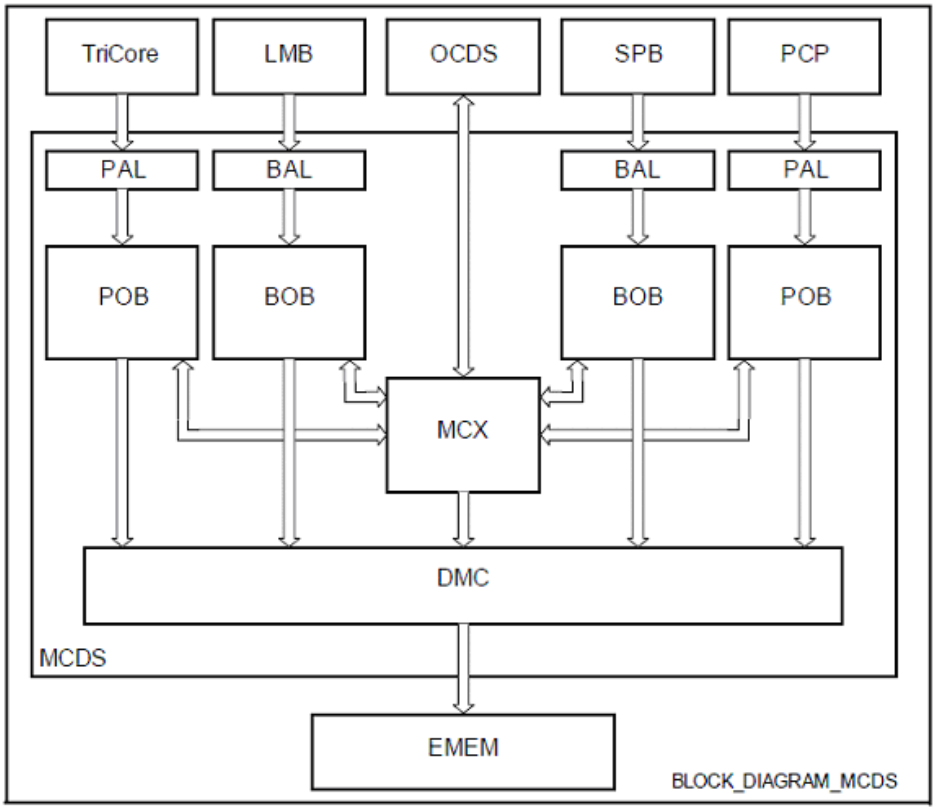
TC1767/TC1797ED block diagram

TC1767/1797ED block diagram shows the Product Chip Part, which is part of the standard product chip and EEC block, which exist on ED device only for the emulation and test needs. The product chip part is reduced to selected modules. The EEC part is implemented on additional silicon area located outside of product chip part and simply left away during manufacturing the product chips.

Most important blocks of the EEC part are MCDS and EMEM. A MLI bridge is used to link the FPI bus of product chip (System Peripheral Bus) and FPI bus of EEC (Back Bone Bus).

The Emulation Memory (EMEM) on the EEC is used for two conceptually different purposes: Calibration and Tracing. Tracing is the focus of this chapter.

Next picture shows the external connections of the MCDS module and its top level internal structure.



TC1767EDMCDS Block diagram

The prime target for the EEC is to cater for the real-time tracing of the TriCore cores. Nonetheless all event generating logic can be used for breakpoint generation via the central break switch as well. It should be noted that there is a certain latency using this route.

TriCore ED trace features:

- TriCore program trace
- TriCore load/store data trace (no register file trace)
- PCP ownership trace
- PCP program trace
- PCP data write to PRAM trace (no register file trace)
- Full visibility of internal peripheral bus (SPB)
- Full visibility of Local Memory Bus (LMB)

9.1.1 Trace Memory

While there are more local FIFOs inside the MCDS to overcome the local bandwidth bottlenecks when more messages are generated at the same time, the part of the Emulation Memory (EMEM) allocated to tracing acts as the actual trace buffer. Up to 1MB (size depends on the emulation device) of EMEM can be allocated for tracing as a buffer memory, which stores the trace messages coming out of the trace unit. Valid data in the buffer is uploaded by the external debugger, which reconstructs the original program flow.

Full Emulation Memory can be also used for other purposes but tracing such as calibration, code, constants or data storage. When full Emulation Memory is used for other purposes, trace and functionalities based on trace such as profiler and coverage cannot be used. However, available Emulation Memory tiles can be split between different use purposes. Amount of Emulation Memory, which target application doesn't

use, respectively makes available for tracing, must be specified in the Hardware/Emulation Options/CPU Setup/Advanced Options (see chapter 3.4 for more details). More Emulation Memory is allocated for the trace buffer, the longer trace, profiler and coverage session will be. Be cautious when Emulation Memory is being used by the application. If the target application would inadvertently use the same tiles of Emulation Memory, which were allocated for tracing, trace could behave unexpectedly or would not work at all.

In case of program trace, trace buffer stores only information of non-sequential instructions being executed. All sequential instructions between the non-sequential instructions are reconstructed by the debugger relying on the correctness of the code information extracted from the debug download file(s). For this reason, tracing of a self-modifying code is not possible. An example of the self-modifying code would be a boot-loader, which copies some code in the internal RAM, where it's then executed. This code cannot be traced as long as its image is not available in the debug download file.

No considerable compression is possible for Data and Ownership trace due to the nature of the data, which needs to be captured. Depending on the traffic of the Data and Ownership messages, the trace buffer can run out relatively quickly. To get the most out the trace memory, Infineon implemented complex qualification and trigger units, which allows capturing respectively filtering only the information of interest. It's up to the user to set up an optimal filter in order to capture only the important information out of the vast amount of information, which PTU, DTU and OTU provide.

9.1.2 Multi-Core Debug Solution (MCDS)

Note: Following description of the MCDS is valid for the TC1767ED device. Other emulation devices (e.g. TC277TE) have obvious differences within the MCDS while the concept remains the same. In-depth understanding of the trace requires studying Target Specification document for the specific emulation device.

Major four blocks of the TC1767ED MCDS are:

- ***Processor Observation Block (POB)***

  Each processor to be traced is paired with a dedicated Processor Observation Block (POB).

  Observation Block for TriCore
  • Direct access to the 4 IP/EA pretrigger comparators of OCDS L1
  • 6 additional range comparators on the IP (this means 6 different ranges)
  • 4 range comparators on the LMB write address
  • 4 masked range comparators on the data written to the LMB
  • Thread awareness: 2 comparators to restrict tracing to certain threads only
  • Full non-cached data access visibility with trace at LMB and FPI bus
  • Watch point traces based on all before mentioned comparators
  • Debug status message based on execution mode of core
  • Complete program trace
  • Data trace for Write Back to LMB
  • Dedicated programmable trace enable generator for each trace unit, using all local comparators as potential sources

  Observation Block for PCP
  • 4 range comparators on the IP
  • 4 range comparators on the PRAM write address
  • 2 masked range comparators on the data written to the PRAM

Note: FPI accesses are handled by the SPB below
• Watch point traces based on all before mentioned comparators
• Debug status message based on execution mode of core
• Complete Program Trace
• Data trace for Write Back to PRAM
• Ownership trace based on interrupt level of executed channel program
• Dedicated programmable trace enable generator for each trace unit, using all local comparators as potential sources

- ***Bus Observation Block (BOB)***

  Each multi master on chip bus (LMB/FPI) has its own Bus Observation Block (BOB) to provide the required visibility.

  Observation Block for System Peripheral Bus (SPB)
  • 4 range comparators on the FPI address
  • 4 masked range comparators on the FPI data bus
  • 4 masked range comparators on the FPI operation code/mastership
  Note: These features lock out the OCDS Level 1 features of the SBCU (breakpoints)
  • Watch point traces based on all before mentioned comparators
  • Ownership trace derived from bus arbiter
  • Data trace with or without address information for Read and/or Write access
  • Dedicated programmable trace enable generator for each trace unit, using all local comparators as potential sources

  Observation Block for TriCore LMB Bus
  • Same features as SPB

- ***Multi Core Cross-Connect (MCX)***

  The main challenge with multi core system debugging is keeping a consistent view of the system, the components of which run independently and concurrently. This is achieved by time stamping the buffered trace messages. On the other hand, trace qualification is needed across core boundaries: It may be useful to trace a bus only if a certain processor core executes a specific subroutine. To keep the interfaces minimal, all such flow of information is routed through this central block, named Multi Core Cross-Connect for this reason.

  Multi Core Cross-Connect
  • Gets 4 programmable pretrigger signals from each observation block
  • One dedicated global trace enable for each trace unit
  • 16 universal 16 bit counters, using programmable combinations of pretriggers as count and clear signals
  • Performance counter
  • Limit comparators based on these counters, generating further pretriggers
  • Global trace enables are sums of products (four multi-input ANDs ORed together)
  • Global break generation based on all available pretriggers - including the counters'
  • Bidirectional interface to the break switch on the product chip
  • Global time stamp messages, based on emulation or reference clock.

- ***Debug Memory Controller (DMC)***

  As explained elsewhere, there is little hope to transport all messages from target to host at the very instant of their creation. By integrating substantial amounts of FIFO-organized memory into the target's package it becomes possible to endure bursts of trace messages without loss of information, even if the physical interface is not extraordinary fast. To make the most of the memory an efficient Message Packer is provided to sort the trace messages from the different sources into the RAM. Additionally time markers can

be inserted by the DMC which allow a reconstruction with accuracy down to the emulation clock cycle - if the memory required for the markers is sacrificed.

MCDS building blocks (generic modules) in a bottom up manner:

- Trigger Logic
- Event Logic

  When combining trigger results, two main cases are possible: Either the event is given by a number of triggers which have to match concurrently (e.g. address in range AND data equal to value) or something has to happen if at least one from a set of triggers (e.g. address lower than bottom OR higher than top of stack) matches.
  As the second case very often requires ANDing triggers to formulate the elements of the OR set, it was decided to relegate the OR function to the event's consumers, namely the Action Definitions.
  Number and kind of triggers connected to each event depend on the location of its implementation.

- Sequential Event Logic

  In some applications an event is defined to have occurred when some triggers have happened in a certain or even arbitrary sequence. Key to the implementation for this problem is the concept of counting events and comparing the count values to given limits. The result of the comparison (cnt_trig) is then treated as a trigger again.

- Performance Counter

  The available counter structure can also be used for performance analysis.

- Action Logic

  All activity inside MCDS is controlled by standardized registers called action definitions.

- Program Trace Unit (PTU)

  The generic PTU is able to process the instruction pointer of an arbitrary processor core.

- Data Trace Unit (DTU)

  The generic DTU is able to process transactions on an arbitrary bus system, consisting of address, data and control information

- Ownership trace unit (OUT)

  Ownership is a NEXUS term; it translates to "Task ID" or "Process ID" for most practical purposes. The generic OTU is able to process the ownership information of an arbitrary processor core if implemented in hardware.

- Watch-point Trace Unit (WTU)

  All comparators are implemented inside trace units (PTU, DTU, OTU). To keep the trace units simple, each of them is allowed to produce up to one trace message per clock cycle only. The watch-point trace messages are produced here to resolve the obvious problem of a watch-point and the "normal" trace being requested at the same time.

- Trace Qualifier Unit (TQU)

  The TQU is the container for all Event and Action Logic. The calculation of the trace qualification signals needed by the trace units (DCU, PTU, DTU, OTU, WTU and TSU) is centralized in a unit of this kind.

- Time Stamp Unit (TSU)

  This block delivers all time information, both for internal use (time tags) and messages (time stamps), to all the other parts of MCDS.

Note: A detail explanation of the TriCore trace is beyond the scope of this document. Contact your local Infineon representative if you need more details.
Due to the complexity of the TriCore trace, iSYSTEM tools offer easy to use Trace Wizard, which makes easy to set up most common trigger and qualifier scenarios (trigger & qualifier on program counter and trigger & qualifier on data).

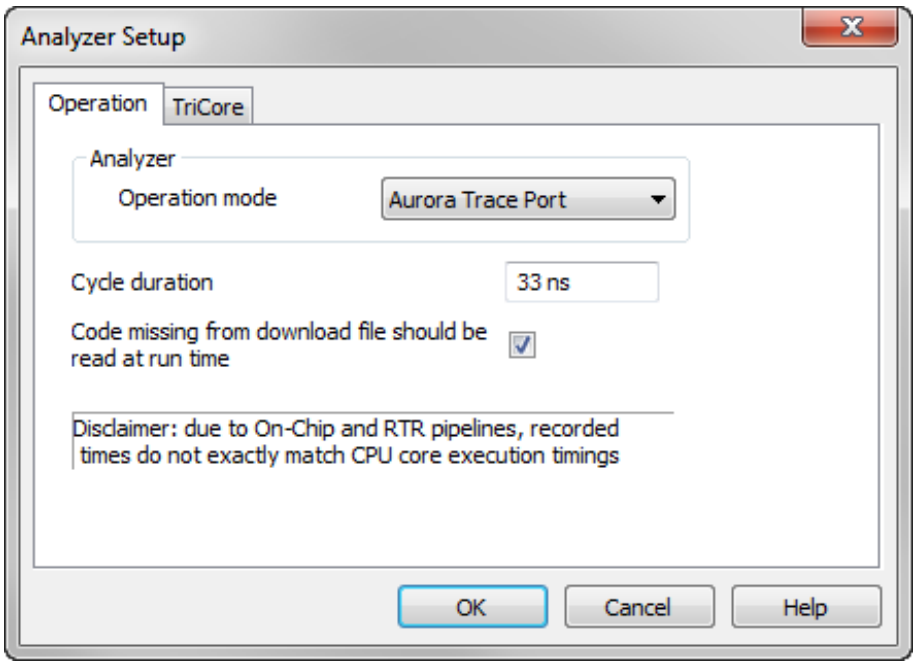
9.2 Aurora Trace Port

Some Tricore emulation devices (ED) provide Aurora trace port (physical interface), which is supported by iSYSTEM iC6000 On-Chip Analyzer including Aurora protocol support.

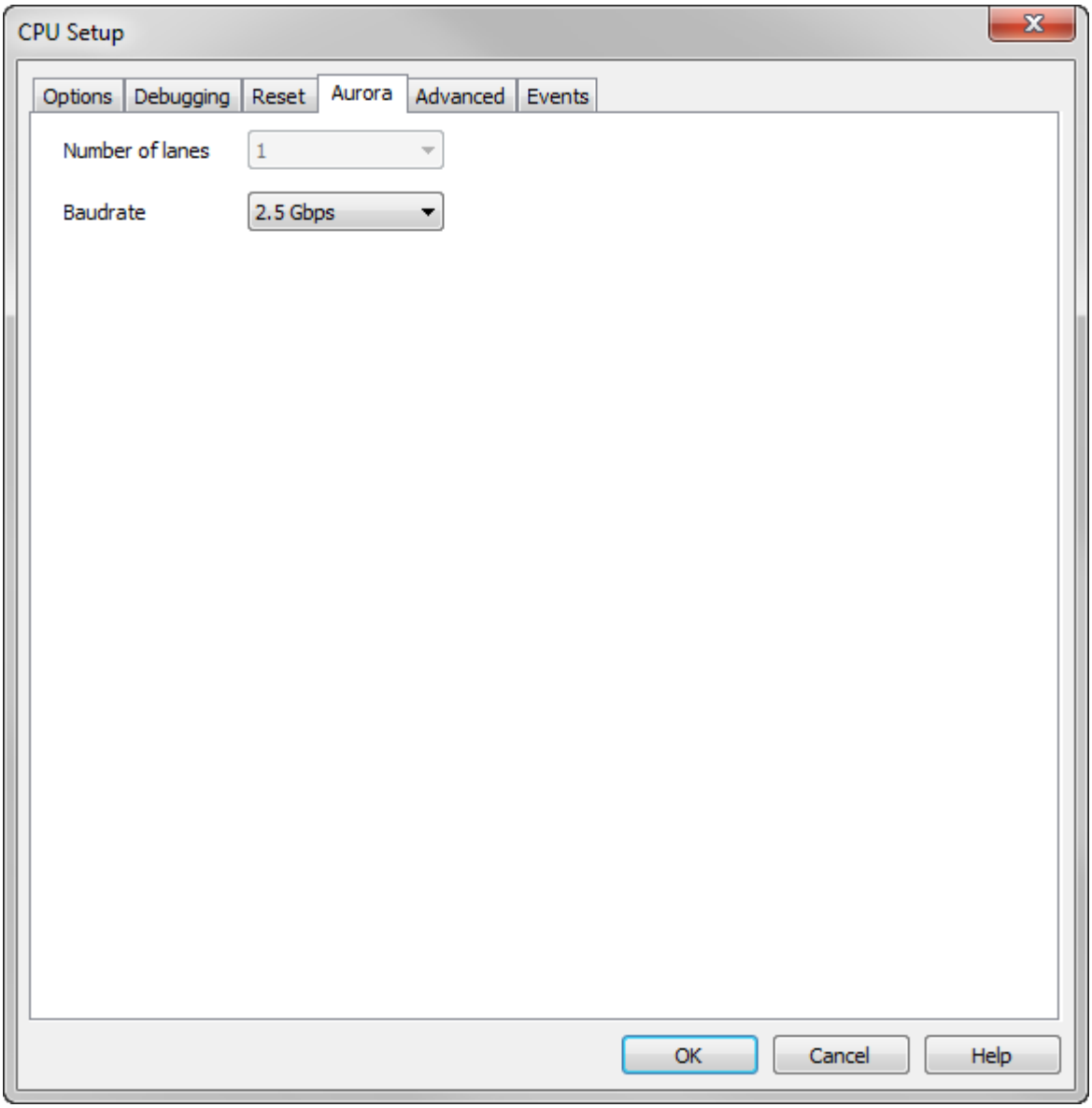
A high-speed Aurora interface from Xilinx is being used as a trace port on high-end (typically multi-core) microcontrollers, where previous trace port technology could no longer keep up with increased trace data bandwidth requirement. Note that a dedicated Aurora debug connector is provided on the target.

Some emulation devices can broadcast on-chip trace information either to a dedicated on-chip trace buffer or to an external debug tool over the Aurora interface.

Select ‘Aurora Trace Port’ in the Hardware / Analyzer Setup dialog for Aurora interface use and select ‘On-chip’ for on-chip trace buffer use.



Aurora specific settings are available in the ‘Aurora’ tab in the ‘Hardware/Emulation Options/CPU/CPU Setup’ dialog. Number of Aurora lanes and a baudrate (speed) are configurable.



Hardware / CPU Setup dialog with Aurora settings

Aurora Trace trigger

Aurora trace port features a USER_IO pin (pin 18 on the 22-pin AGBT connector), which is used by the debugger to receive the trigger signal. Trigger output signal (TRO) needs to be routed from the MCDS to this pin by means of configuring the MCU.

In order to route the trigger output to the aurora trace port, use an initialization script such as the one below, which routes trigger output signal through the MCU P32.6 pin. Note that the script must be adjusted if other MCU pin is physically connected to the USER_IO Aurora trace port pin.

```
// TRACE TRIGGER OUTPUT
A CBS_TOPR  L  0x00040000 // TL4 connected to trigger out pin 4
                          // port P32.6
A CBS_TRMT  L  0x00000004 // MCDS trigger out 0 connected to TL4
A CBS_TOPPS L  0x00000200 // trigger output pulse stretched
                          // to minimum 4PBs (max)
```

```
A P32_PDR0  L  0x30333333 // port P32.6 - pad driver characteristic
                          // set to speed grade 1 (max)

// DISABLE TRACE TIME WHEN CPU IS STOPPED
// Master CPU (CPU0) connects HALT output to TL2
// MCDS break_in connection
A CBS_TRMC  L  0x00200000 //MCDS Break in is connected to TL2
```

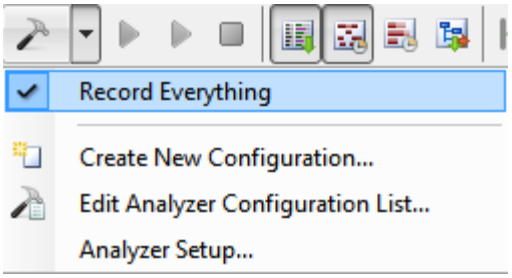
9.3 Trace Configuration

Analyzer window is open from the View menu.

Before using trace for the first time please refer to winIDEA Contents Help, Analyzer Window section. Configuration windows may differ slightly from the screenshots presented in the following chapters, as different TriCore devices provide different functionalities.

9.3.1 Record everything

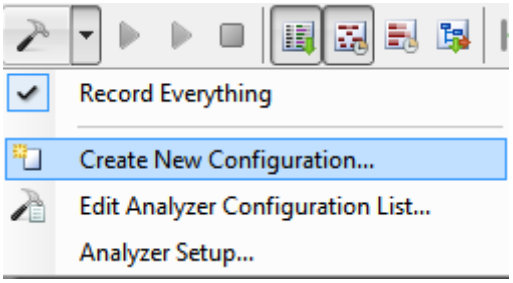
This configuration is provided by default when new trace file is opened and is used to record application program flow from the application run on.



9.3.2 Use Trigger/Qualifier

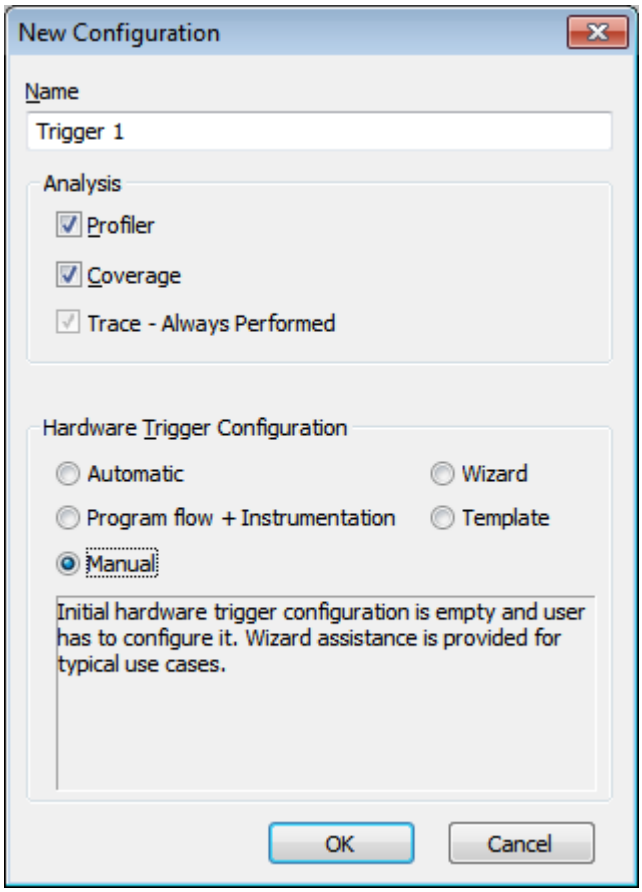
This trace operation mode is used, when it's required to trace the application around a particular event or when only some parts of program or data have to be recorded. In practice it turns out to be the most important matter defining meaningful trace qualifiers.

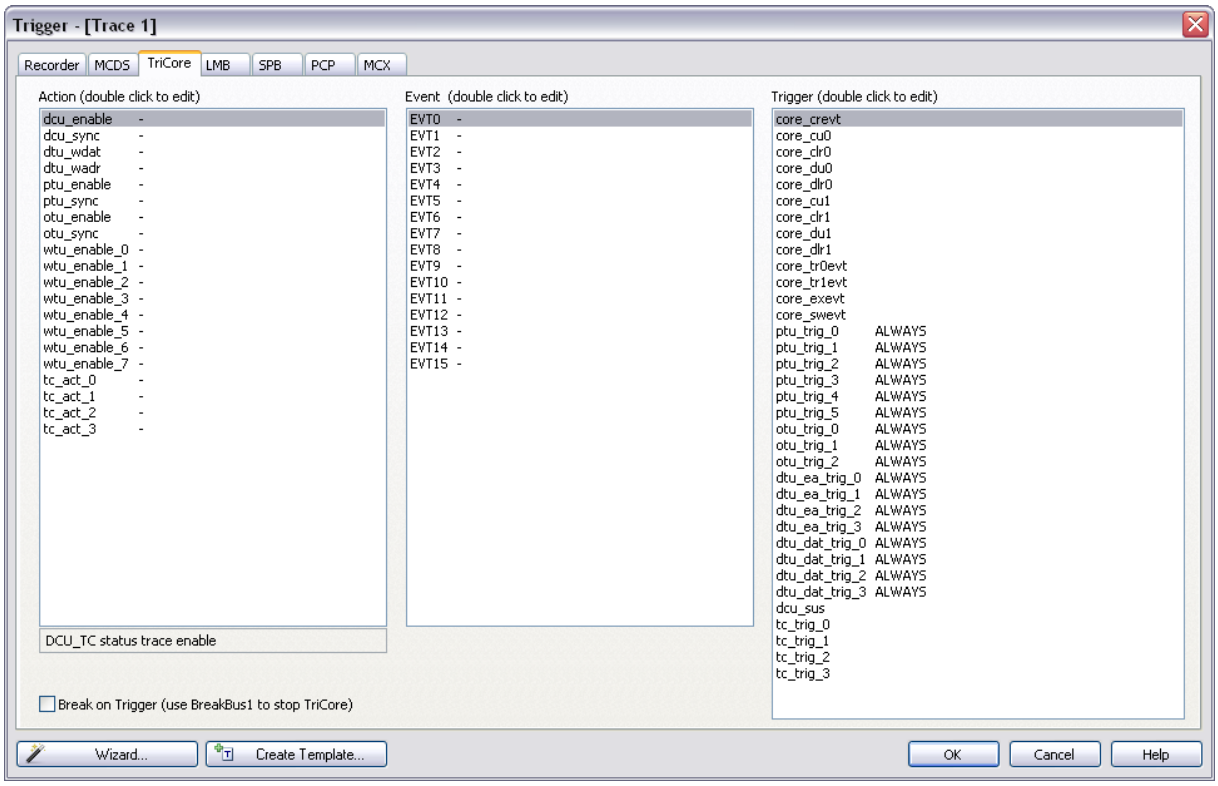
Select 'Create New Configuration' in the trace window,



Next select 'Manual' under Hardware Trigger Configuration in the newly opened New Configuration dialog.

When configuring trace for default profiler or coverage operation, 'Automatic' should be selected. 'Wizard' selection provides simple to use GUI and eases configuring trace trigger e.g. on a function call or on a simple data write access.





TriCore Trace Qualifier Unit configuration

TriCore, LMB, SPB and PCP have each its own Trace Qualifier Unit (TQU) and hence its own pool of possible:

- triggers (right section in the dialog)

  These are also referred as pre-triggers. Pre-trigger is a system state (status bit or comparator output).

- events (centre section)

  An event is the combination of pre-triggers at a certain point of time.

- and actions (left section)

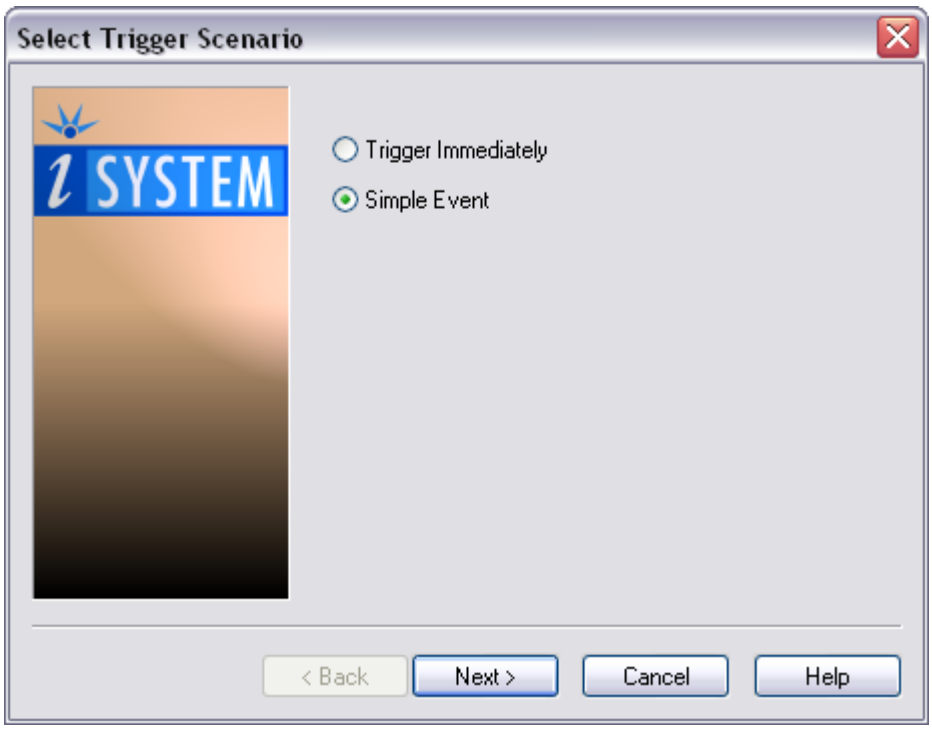
  An action is something to be done when an event happens.

The MCX is not paired with a specific core or bus, but interfaces to the TQUs of all observation blocks (TriCore, LMB, SPB and PCP). A central trace qualification unit (TQU) contains the bulk of its functionality. Additionally the central time base (TSU), the usual watch-point message generation (WTU) and message sequencer (MSU) building blocks are implemented.

It is recommended to use iSYSTEM Trace Wizard when it's required to:

- set a simple trigger on an executed function
- set a simple trigger on a single data access
- trace only one program range
- trace only data access to a single address range

iSYSTEM Trace Wizard is invoked in the left bottom corner in the Trigger dialog and it configures all the necessary trace qualification units accordingly. When there is a demand for more complex trigger(s) and/or qualifier(s), the user should engage himself in the configuration of individual trace qualification unit.



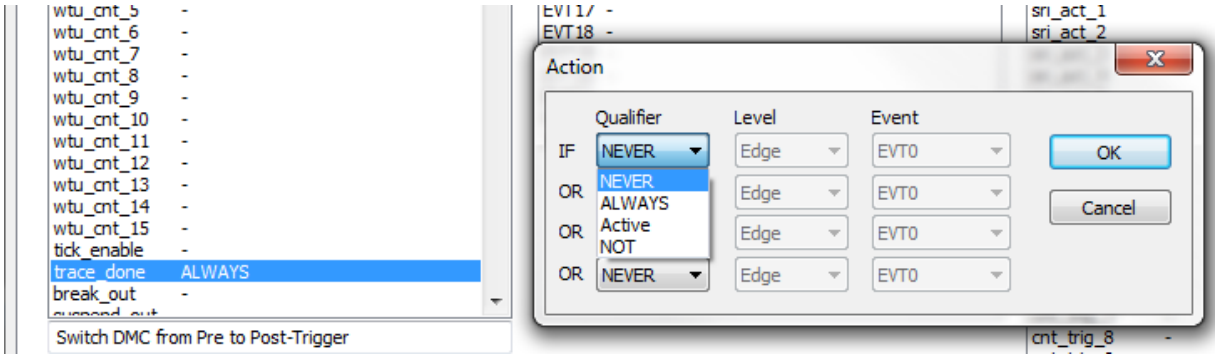
First page of the iSYSTEM Trace Wizard

9.3.3 Continuous trace recording using upload while sampling (UWS)

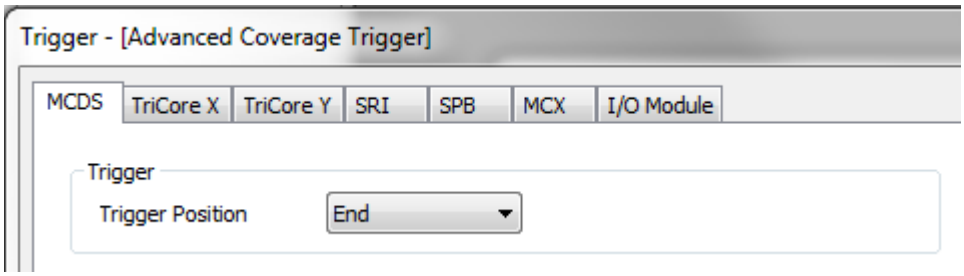
Continuous trace is used, when we are interested in analyzing the application in an extended time frame. In this mode debugger waits for trace to finish recording the EMEM tile and then uploads it. In the meantime TriCore is storing data in the following tiles. This in theory allows for a continuous trace recording. However, recording may be faster than the upload, in which case trace is stopped when TriCore tries to store data in the tile, which is currently being uploaded.

To avoid this situation, use a trace qualifier, which will produce small-enough flow of data to the EMEM.

Continuous trace is enabled in the recorder configuration by setting trace_done action in the MCX tab to NEVER, as seen on the image below and set trigger position to End, which can be done in the MCDS tab (see images below).



Setting the trace_done action to NEVER



Setting the trigger position to End

9.3.4 Trace over HW reset

When we are interested in program flow during HW reset, 'Enable trace during CPU reset' option in MCDS tab in the recorder configuration must be enabled. This is necessary, because the trace buffer is cyclic and trace registers are reset together with the CPU, so it is impossible to determine where the trace recording has stopped (which message was first and which is last in EMEM RAM). With 'Enable trace during CPU reset' option checked, winIDEA uses time stamp information (counters stored in the trace buffer) to find the first and the last message.

9.3.5 Trace over SW reset

During SW reset trace is working normally. Everything what is specified as qualifier is stored to trace RAM.

To trace operation before and after reset you can either:

a) Set breakpoint to code which is executed after reset only and select 'Continuous mode' in winIDEA. In this mode trigger (trace_done) is automatically set to never and trigger position to 'end'. After SW reset CPU stops at breakpoint, trace is stopped and history is visible.

b) Set trace trigger (trace_done) to function which is executed after reset only and trigger position to 'end'. After SW reset trace trigger occurs, trace is stopped and history is visible.

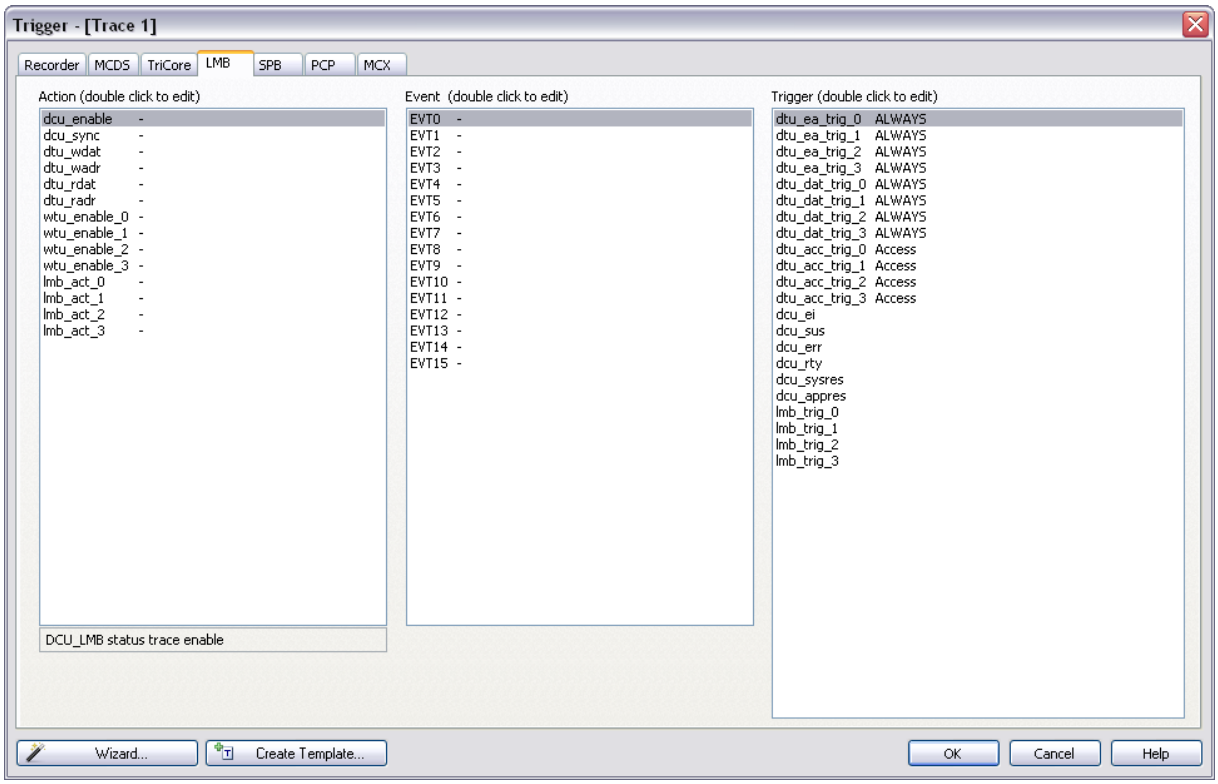
SW reset can take a while so timestamp messages can fill onchip trace memory RAM completely. Disable time stamps or decrease the frequency of time stamps (if tsu_rel is used increase the value of the TSUPRSCL)

9.3.6 TriCore Trace Qualifier Unit (TQU_TC)

Feature Overview

- Dedicated programmable trace enables for each Trace Unit of POB_TC.
- Four dedicated actions for signaling to the TQU_MCX.
- 13 core triggers from the OCDS-logic of the TriCore.
- Six range triggers from the PTU_TC.
- Four range triggers on the write-back address seen by DTU_TC.
- Four masked and signed data triggers from the DTU_TC.
- Three triggers from the OTU_TC.
- One trigger from the DCU_TC.
- Four triggers from the TQU_MCX.
- Five uncommitted trace enables from the TQU_MCX.

9.3.7 Local Memory Bus Trace Qualifier Unit (TQU_LMB)

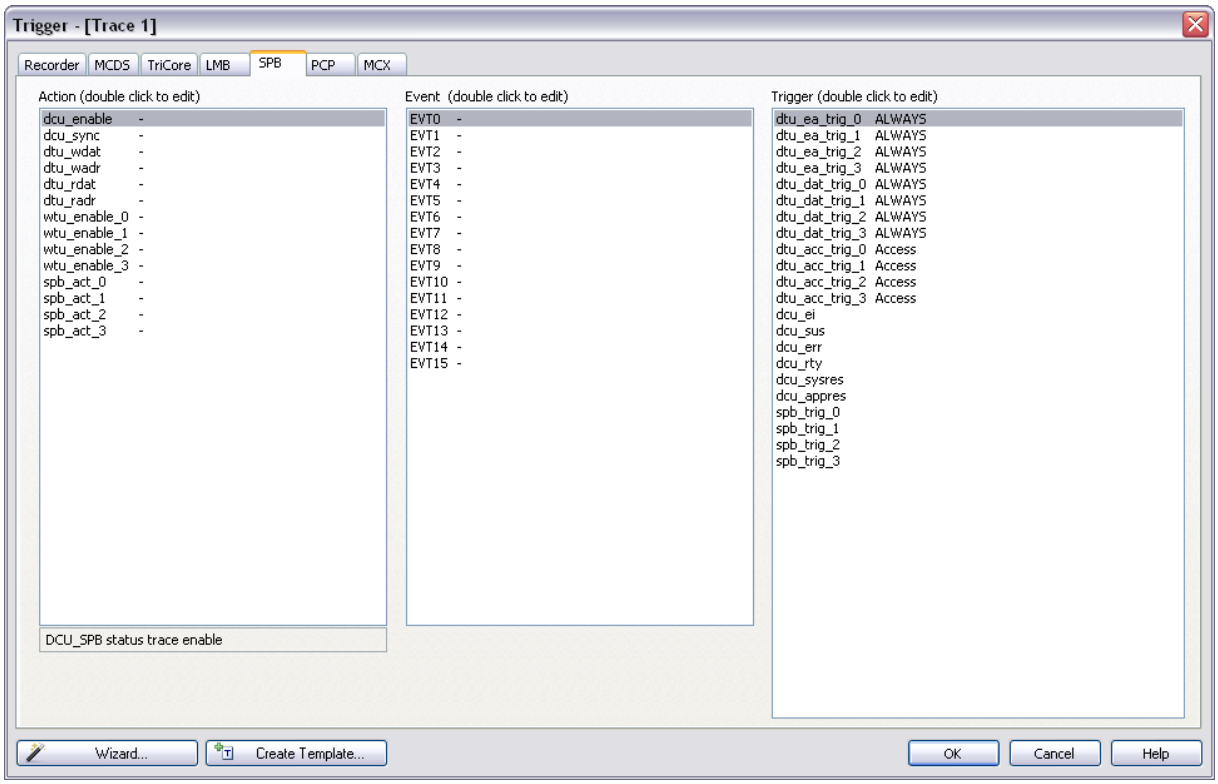


LMB Trace Qualifier Unit configuration

Feature Overview

- Dedicated trace enables for each Trace Unit of BOB_LMB.
- Four dedicated actions for signaling to the TQU_MCX.
- Four range triggers on the transaction address seen by DTU_LMB.
- Four masked and signed data triggers from the DTU_LMB.
- Four range triggers on the transaction type and bus master seen by DTU_LMB.
- Six triggers from the DCU_LMB.
- Four triggers from the TQU_MCX.
- Five uncommitted trace enables from the TQU_MCX.

9.3.8 System Peripheral Bus Trace Qualifier Unit (TQU_SPB)



SPB Trace Qualifier Unit configuration

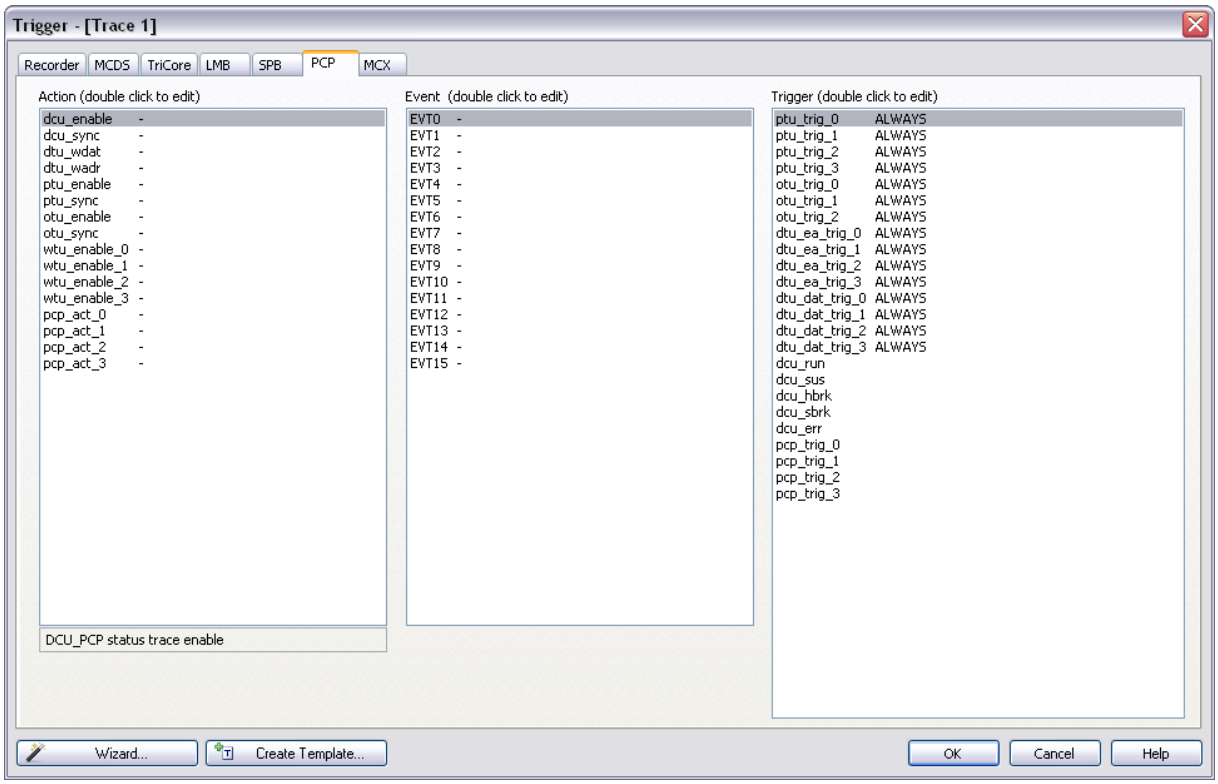
Feature Overview

- Dedicated trace enables for each Trace Unit of BOB_SPB.
- Four dedicated actions for signaling to the TQU_MCX.
- Four range triggers on the transaction address seen by DTU_SPB.
- Four masked and signed data triggers from the DTU_SPB.
- Four range triggers on the transaction type and bus master seen by DTU_SPB.
- Six triggers from the DCU_SPB.
- Four triggers from the TQU_MCX.
- Five uncommitted trace enables from the TQU_MCX.

9.3.9 PCP Trace Qualifier Unit (TQU_PCP)

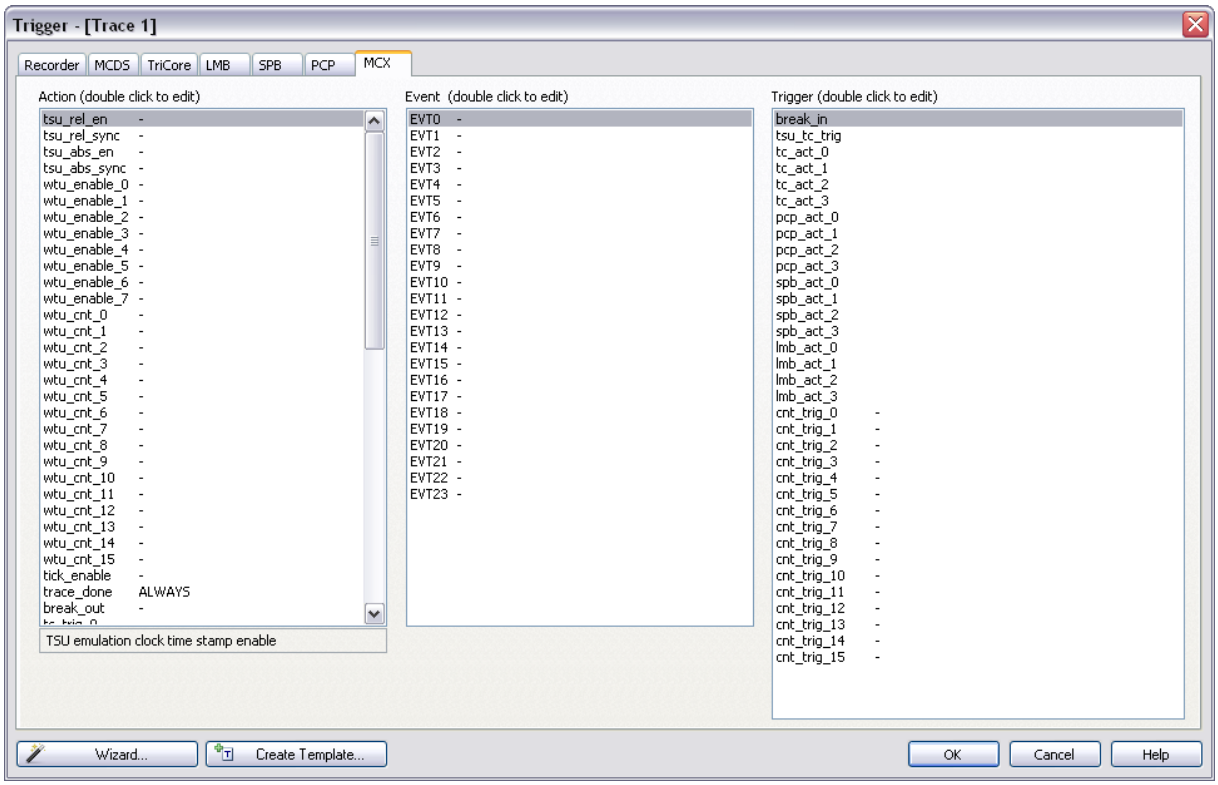
Feature Overview

- Dedicated trace enables for each Trace Unit of POB_PCP.
- Four dedicated actions for signaling to the TQU_MCX.
- Four range triggers from the PTU_PCP.
- Four range triggers from the PRAM write address seen by DTU_PCP.
- Four masked and signed data triggers from the DTU_PCP.
- Three triggers from the OTU_PCP.
- Five triggers from the DCU_PCP.
- Four triggers from the TQU_MCX.
- Five uncommitted trace enables from the TQU_MCX.



PCP Trace Qualifier Unit configuration

9.3.10 Central Trace Qualifier Unit (TQU_MCX)



MCX Trace Qualifier Unit configuration

Note that only triggers cnt_trig0 to cnt_trig15 are configurable on the right. Other triggers are actions already from the TriCore, LMB, SPB and PCP observation blocks. MCX Trace Qualifier Unit configuration dialog holds also the time stamp settings.

Feature Overview

• Dedicated trace enables for each Trace Unit of MCX.

• 16 universal 16 bit counters, using programmable combinations of triggers as count and clear signals.

• Programmable limit comparator in each counter.

• Passing a limit is available as unique trigger for each counter.

• The counter values can be traced (see WTU_MCX).

• Pre-scaled reference clock available as trigger.

• Four triggers from each observation block's TQU.

• Four triggers to each observation block's TQU.

• Five uncommitted trace enables to each observation block's TQU.

• Trigger from OCDS break switch.

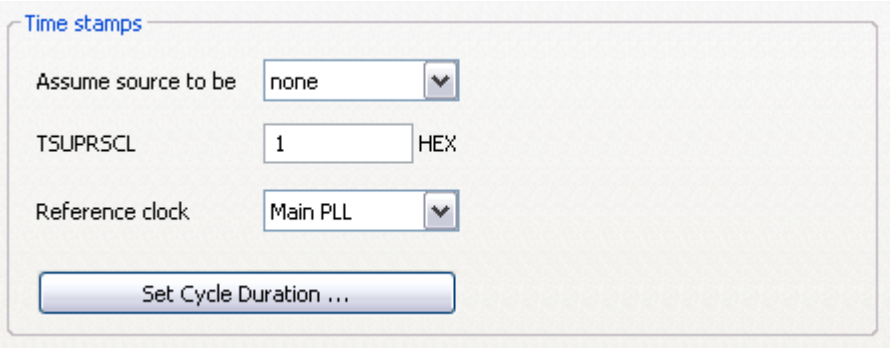
• Trigger to OCDS break switch.

• 39 performance signals from TriCore, Flash, LMB, PCP, DMA, SPB.

9.3.11 Time Stamp Configuration

By default, MCDS does not generate time stamp messages, which is somewhat distinct to Nexus trace implementations on other architectures. This means, in order to get the time stamp information as part of the trace record, the MCDS needs to be configured accordingly. Note that the trace time stamp information is not based directly on the time but is implemented on a CPU (MCDS) tick level. The trace user has to determine the MCDS tick period in order to get the time information in the trace. This method of course relies on the assumption that the CPU clock doesn't change during the active trace session. Time cannot be measured properly when tracing program code changing the CPU clock.

As the time stamp mechanism is relatively complex too, the most convenient configuration will be described.

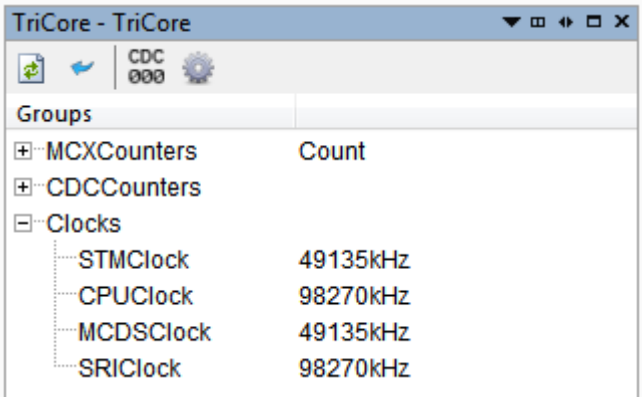


Time Stamps area in the MCDS tab in Trigger dialog

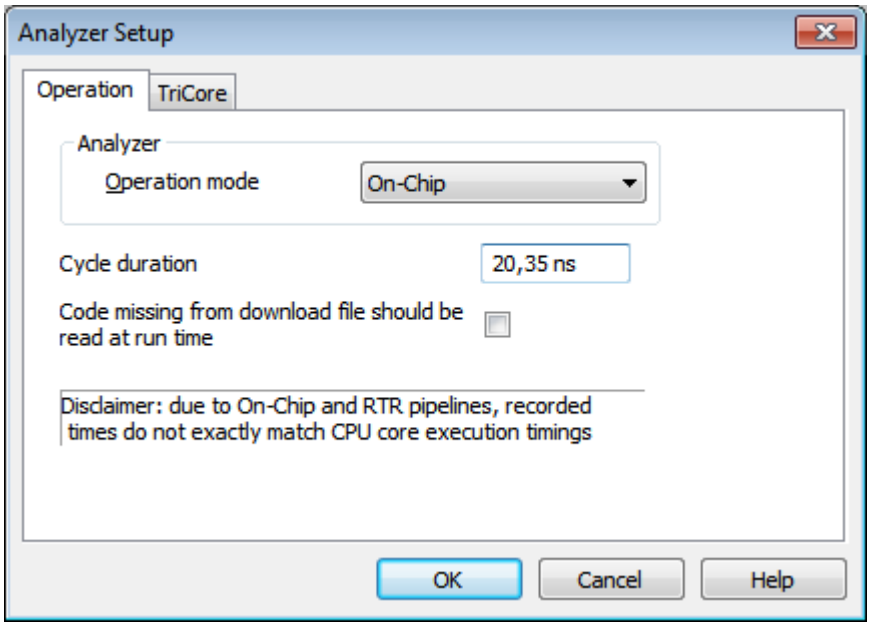
Set 'Assume source to be' to 'tsu_rel', keep default TSUPRSCL set to 1 and Reference clock set to Main PLL. A value of 1 for TSUPRSCL activates the tsu_tc_trig output every second reference clock cycle. Increasing the value would yield higher time resolution.

Next, open the Measurement plug-in window from the 'Plugins/Measurement' menu. Run the application and press the 'Refresh' button in the Measurement plug-in window. Actual MCDS

clock is displayed under Clocks group. Now calculate period for this clock. In this particular case, it's 20.35 ns for 49.135 MHz clock.



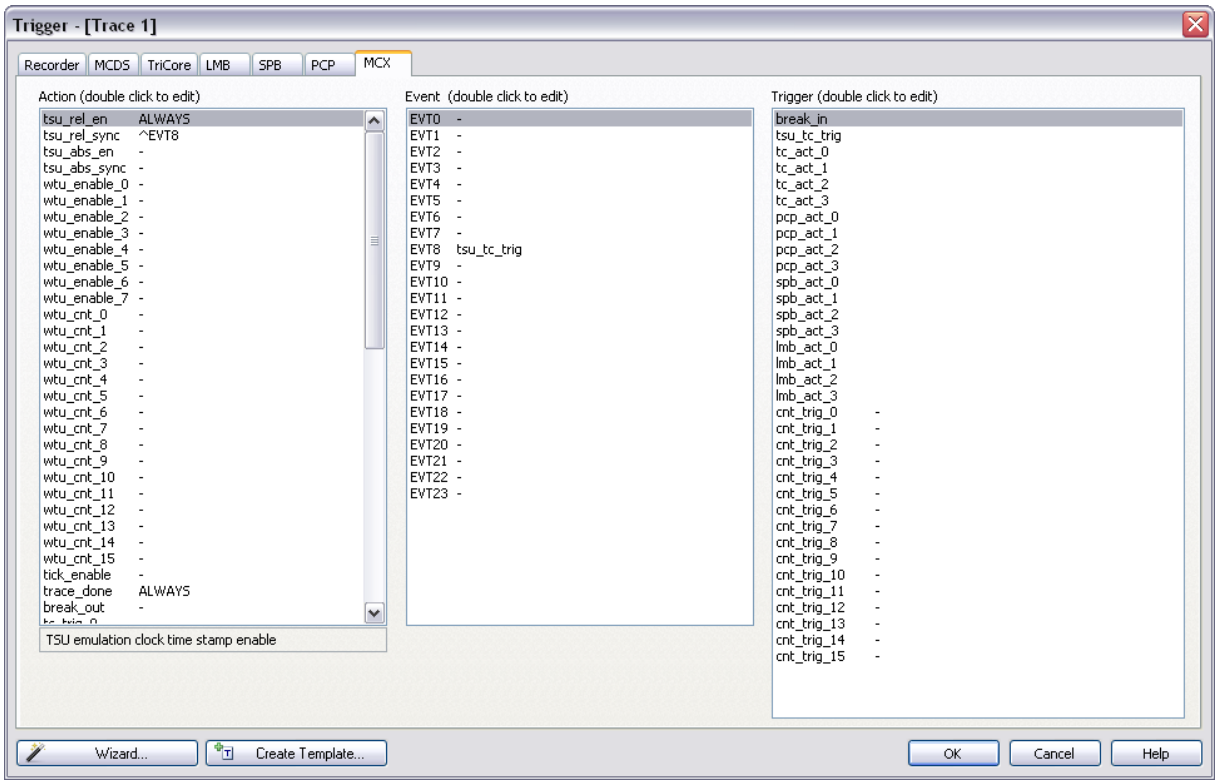
Next, press 'Set Cycle Duration …' button in the Time Stamps area and set period at the bottom of the newly opened dialog.



'Hardware/Analyzer Setup' dialog

While this time stamp configuration is already operational for the trace 'Record everything' operation mode, MCX need to be configured additionally to generate time stamps for the trace 'Trigger/Qualifier' operation mode.

The easiest way is to use iSYSTEM trace wizard, which is started by pressing the 'Wizard…' button in the left bottom corner in the Trigger configuration dialog. Time stamps generation will be configured regardless of the specific trigger and/or qualifier setting. If trigger immediately and record program flow is configured via Wizard, MCX settings look like this:



Time stamps related settings are:

- event EVT8 is set to ‘tsu_tc_trig’ Trigger
- ‘tsu_rel_en’ Action is set on ALWAYS
- and ‘tsu_rel_sync’ is set to EVT8 Event edge

‘trace_done’ Action set to ALWAYS yields recording everything from the trace start on.

Before the trace time stamp information is used, Cycle duration setting must be set in the ‘Hardware/Analyzer Setup’ dialog. The trace time stamp information is implemented on a CPU tick level only. Therefore, it’s up to the user to find out the CPU cycle period of his target application and enter that value here. This type of time stamps doesn’t provide accurate trace time information for applications where the microcontroller frequency is not constant during the trace session.

10 Profiler

Refer to winIDEA Contents Help, Profiler Concepts section for profiler theory and background.

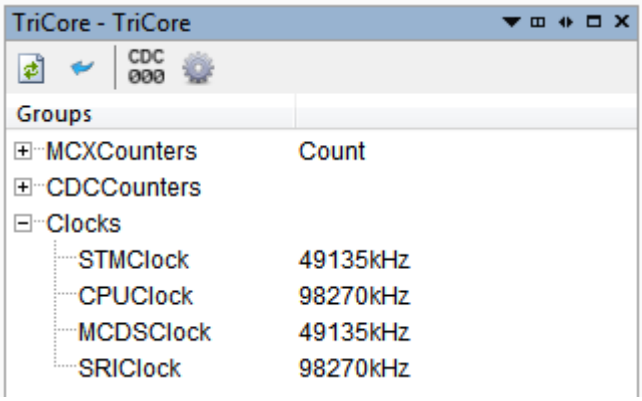
Refer to winIDEA Contents Help, Analyzer Window section (or alternatively to the standalone Analyzer.pdf document) for information on profiler user interface and use.

Note: Profiler is available on TriCore Emulation Devices (ED) only.

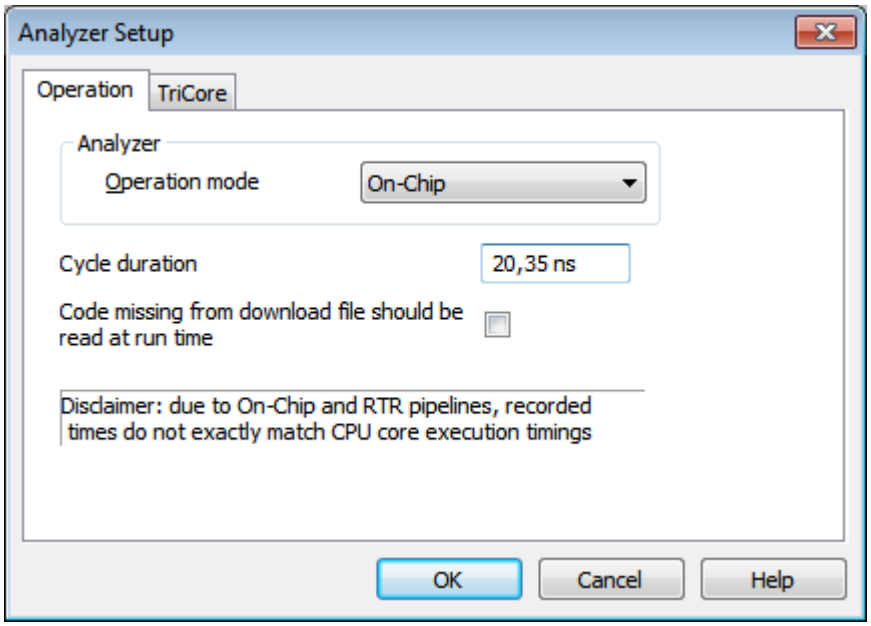
Hardware Configuration

Before the profiler can be used, a 'Cycle duration' setting must be set in the 'Hardware/Analyzer Setup' dialog. Note that the profiler relies on the results captured by the trace, which on TriCore provides detailed time stamp information on a CPU (MCDS) tick level only. It's up to the user to identify the CPU clock period of his target application and to enter this value in the 'Cycle duration' field. Note that the profiler based on this kind of time stamps only works on for applications where the microcontroller clock doesn't change during runtime. Profiler will yield incorrect results when profiling applications changing the operation clock at the runtime.

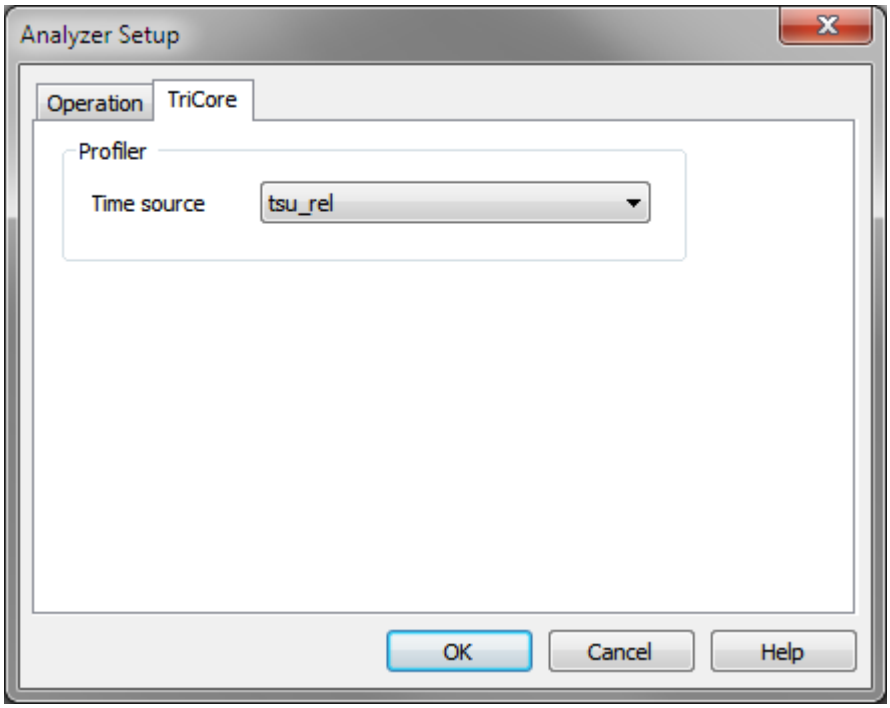
Open the 'Measurement plug-in' window from the 'Plugins/Measurement' menu and run the application. Next, press the Refresh button in the Measurement plug-in window to measure and display the MCDS clock frequency. Now, calculate clock period (T=1/f) for this frequency. In this particular case, it's 20.35 ns for 49,135 MHz clock.



Next, open the 'Hardware/Analyzer Setup' dialog and enter the calculated clock period in the 'Cycle duration' field at the bottom of the dialog.



Keep default ‘Time source’ setting (tsu_rel) in the ‘TriCore’ tab within the Analyzer Setup dialog. Refer to the Time Stamp Configuration section in the Trace chapter for more details on this particular setting.



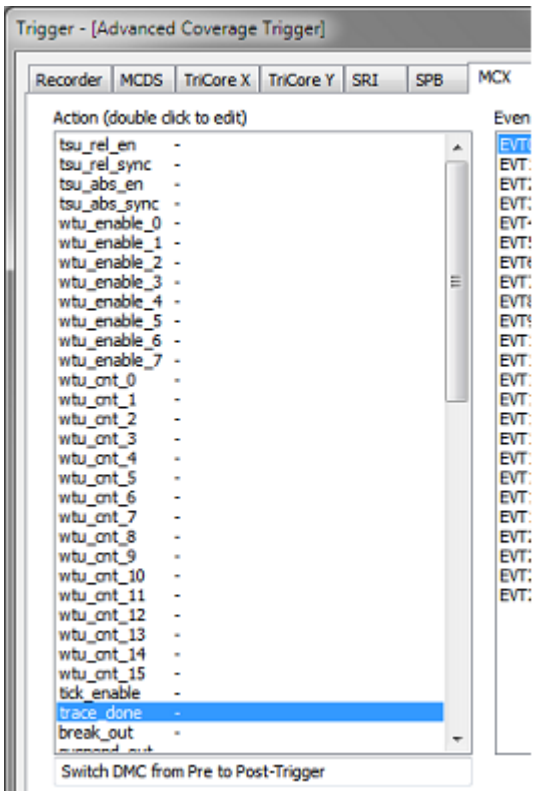
Time related hardware configuration is finished now.

Per default profiler configures on-chip trace logic for recording the complete program flow. In the first place, profiler session time is limited by the physical size of the on-chip trace buffer, which is device specific and can be up to 1MB.

When a limited session time is an obstacle for the test process and not all application functions are required to be profiled, the on-chip trace can be configured to record only functions of interest. Consequentially, this generates less trace messages within the same time frame and the same physical on-chip trace buffer yields longer session time.

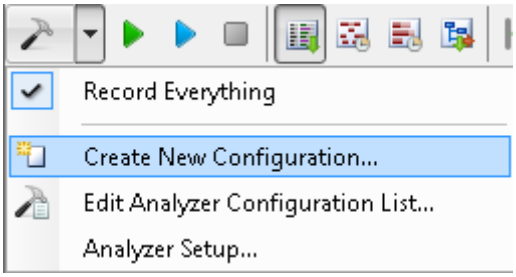
iC5000 debug tool in conjunction with DAP and DAP2 debug interface features a so called “upload while sampling” operation mode, which allows uploading the on-chip trace buffer while the application is still being recorded. When tracing events with low frequency, infinite profiler session is possible.

The ‘trace_done’ action in the MCX tab (trace trigger dialog) set to NEVER is the key setting which configures the on-chip trace logic for ‘upload while sampling’ operation. This operation mode is supported on iC5000 only and when connecting to the target microcontroller through the DAP debug interface. Note that iC5000 must be connected via the USB for a maximum performance.

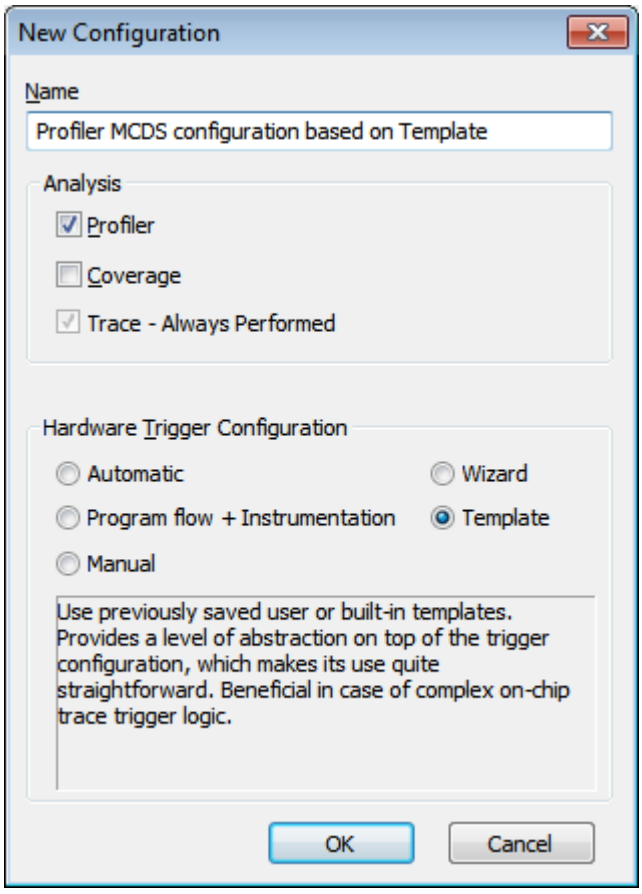


Since Tricore on-chip trace logic is hard to understand and configure, winIDEA provides prebuilt templates, which facilitate configuring on-chip trace logic for e.g. recording one function, 4 functions or a single data object.

Select the ‘Create New Configuration…’ selection available under the ‘Analyzer Configuration’ toolbar.

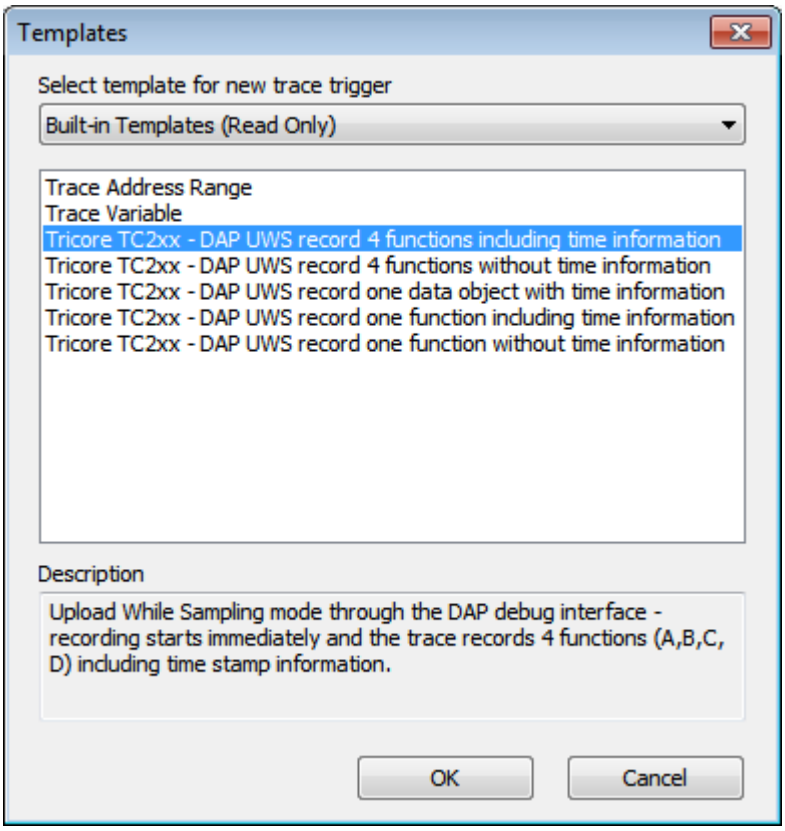


‘New Configuration’ dialog is opened where user can choose several Hardware Trigger Configuration possibilities. ‘Automatic’ and ‘Program flow + Instrumentation’ provide a preconfigured settings. ‘Wizard’ is also a very good starting point since it covers already few typical use cases.

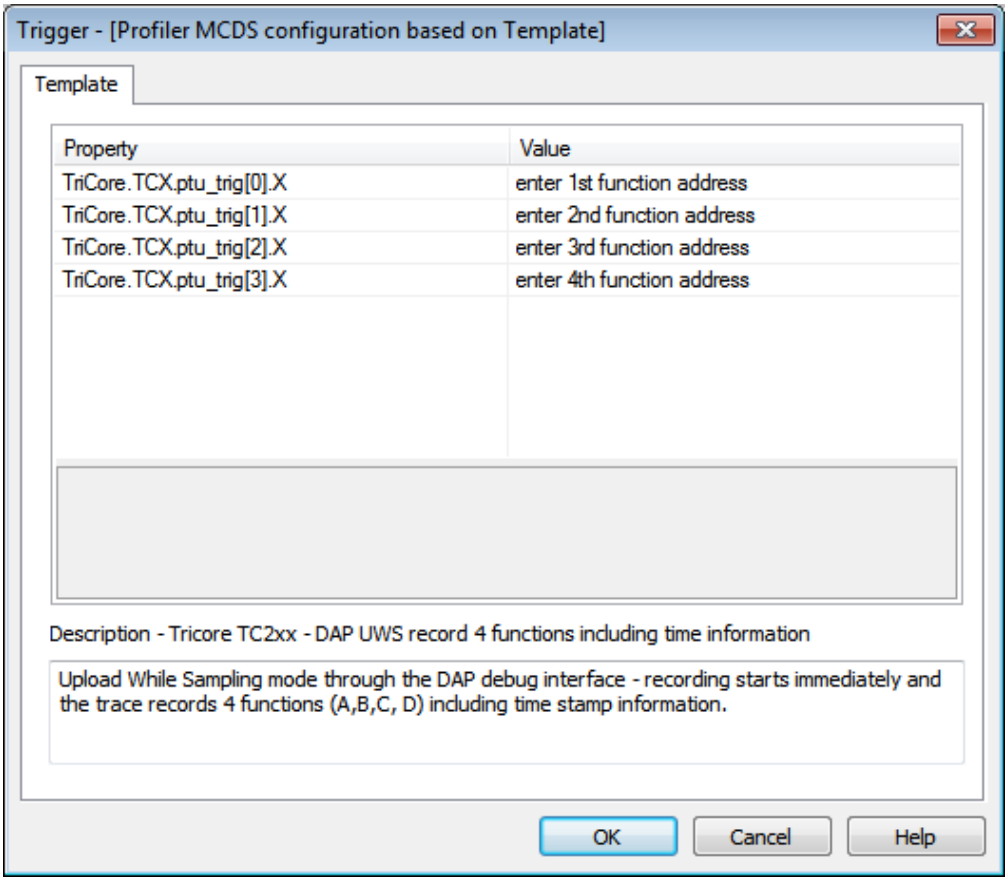


When someone would like to get the most out of the on-chip trigger/qualifier resources, the 'Manual' option should be selected. In this particular case, the 'Template' selection is selected since we will use one of the pre-built templates.

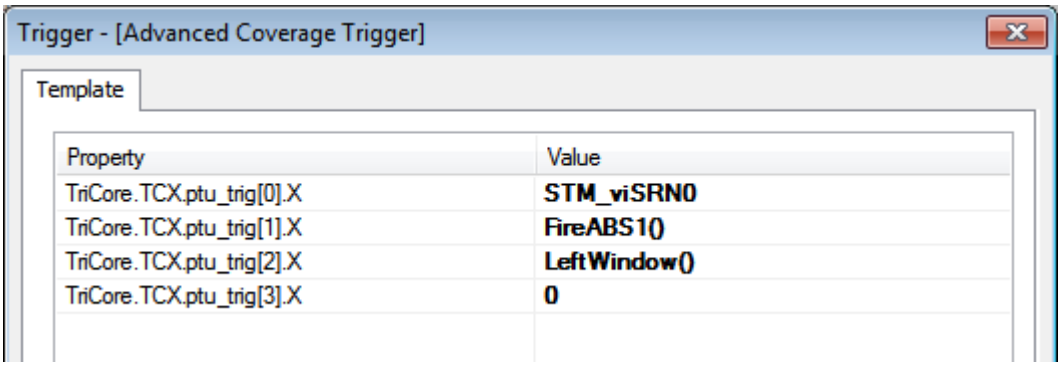
Next, select 'Built-in Templates' in the Templates dialog and pick 'TriCore TC2xx – DAP UWS record 4 functions including time information' prebuilt template.


Templates
Select template for new trace trigger
Built-in Templates (Read Only)
Trace Address Range
Trace Variable
Tricore TC2xx - DAP UWS record 4 functions including time information
Tricore TC2xx - DAP UWS record 4 functions without time information
Tricore TC2xx - DAP UWS record one data object with time information
Tricore TC2xx - DAP UWS record one function including time information
Tricore TC2xx - DAP UWS record one function without time information
Description
Upload While Sampling mode through the DAP debug interface - recording starts immediately and the trace records 4 functions (A,B,C, D) including time stamp information.
OK
Cancel

After confirming the selection, a dialog opens where addresses for 4 functions are entered. If less than four functions need to be profiled, simply enter an address which is never executed (e.g. 0) for the redundant function/code addresses.


Trigger - [Profiler MCDS configuration based on Template]
Template
Property
Value
TriCore.TCX.ptu_trig[0].X
enter 1st function address
TriCore.TCX.ptu_trig[1].X
enter 2nd function address
TriCore.TCX.ptu_trig[2].X
enter 3rd function address
TriCore.TCX.ptu_trig[3].X
enter 4th function address
Description - Tricore TC2xx - DAP UWS record 4 functions including time information
Upload While Sampling mode through the DAP debug interface - recording starts immediately and the trace records 4 functions (A,B,C, D) including time stamp information.
OK
Cancel
Help

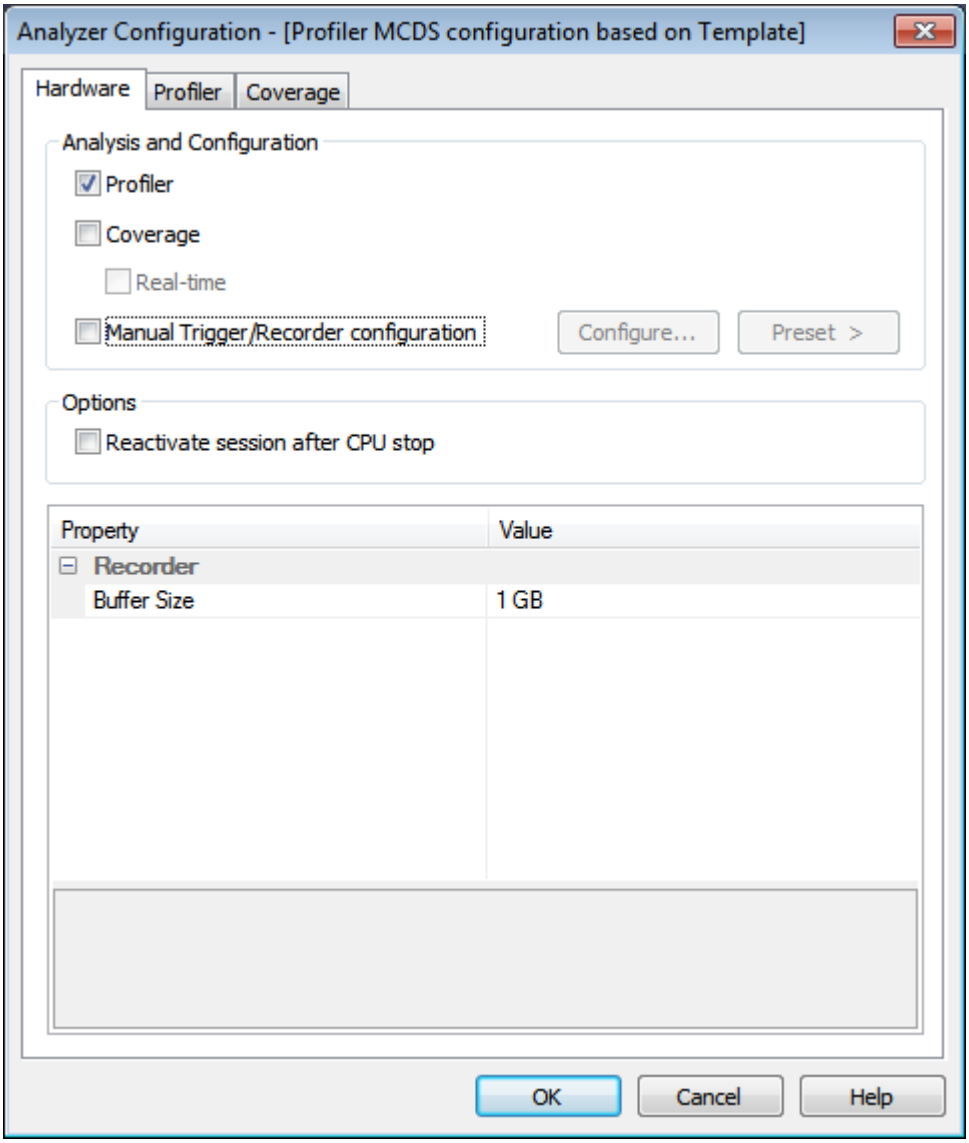
Next picture shows configuration for three functions, which will be recorded and profiled.



Close the configuration and run the profiler.

Note that less functions are profiled and less often they are executed, longer the profiler session will be.

When a complete application must be profiled, uncheck the 'Manual Trigger/Recorder configuration' and the debugger will take care of the necessary on-chip trace configuration.



11 Coverage

Refer to winIDEA Contents Help, Coverage Concepts section for coverage theory and background.

Refer to winIDEA Contents Help, Analyzer Window section (or alternatively to the standalone Analyzer.pdf document) for information on coverage user interface and use.

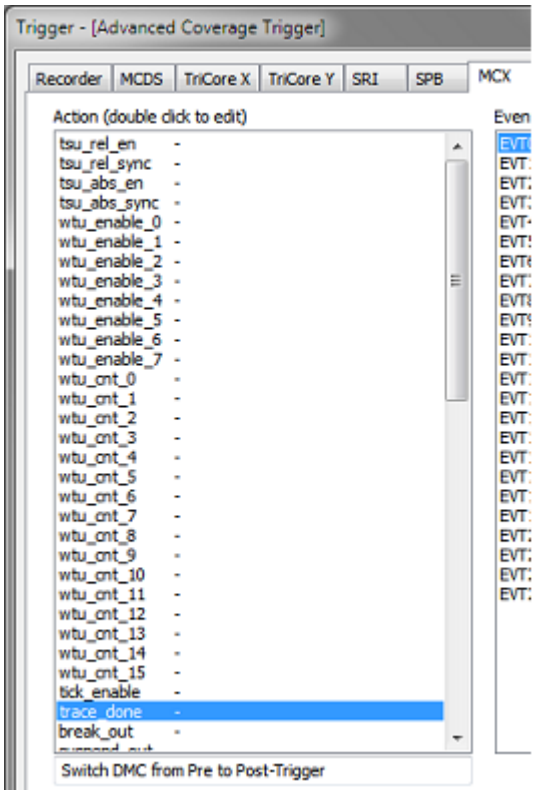
Note: Coverage is available on TriCore Emulation Devices (ED) only.

Per default coverage configures TriCore on-chip trace logic for recording the complete program flow. In the first place, coverage session time is limited by the physical size of the on-chip trace buffer, which is device specific and can be up to 1MB.

When a limited session time is an obstacle for the test process and not all application functions are required to be tested for coverage metrics, the on-chip trace can be configured to record only functions of interest. Consequentially, this generates less trace messages within the same time frame and the same physical on-chip trace buffer yields longer session time. In addition, a time information is irrelevant for the coverage and can be also switched off. This will prolong the session time further more.

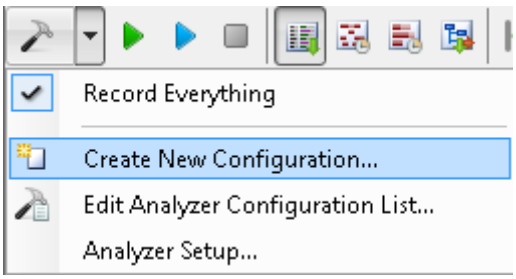
iC5000 debug tool in conjunction with the DAP debug interface features a so called “upload while sampling” operation mode, which allows uploading the on-chip trace buffer while the application is still being recorded that is traced. When tracing events with low frequency, infinite coverage session is possible.

The ‘trace_done’ action in the MCX tab (trace trigger dialog) set to NEVER is the key setting which configures the on-chip trace logic for ‘upload while sampling’ operation. This operation mode is supported on iC5000 only and when connecting to the target microcontroller through the DAP debug interface. Note that iC5000 must be connected via the USB for a maximum performance.



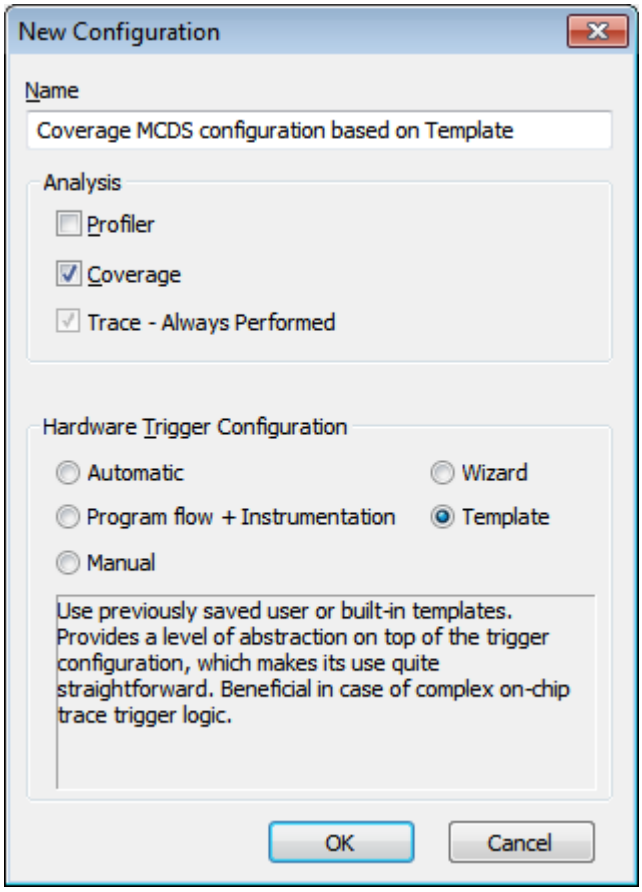
Since TriCore on-chip trace logic is hard to understand and configure, winIDEA provides prebuilt templates, which facilitate configuring on-chip trace logic for e.g. recording one function or 4 functions.

Select the ‘Create New Configuration…’ selection available under the ‘Analyzer Configuration’ toolbar.

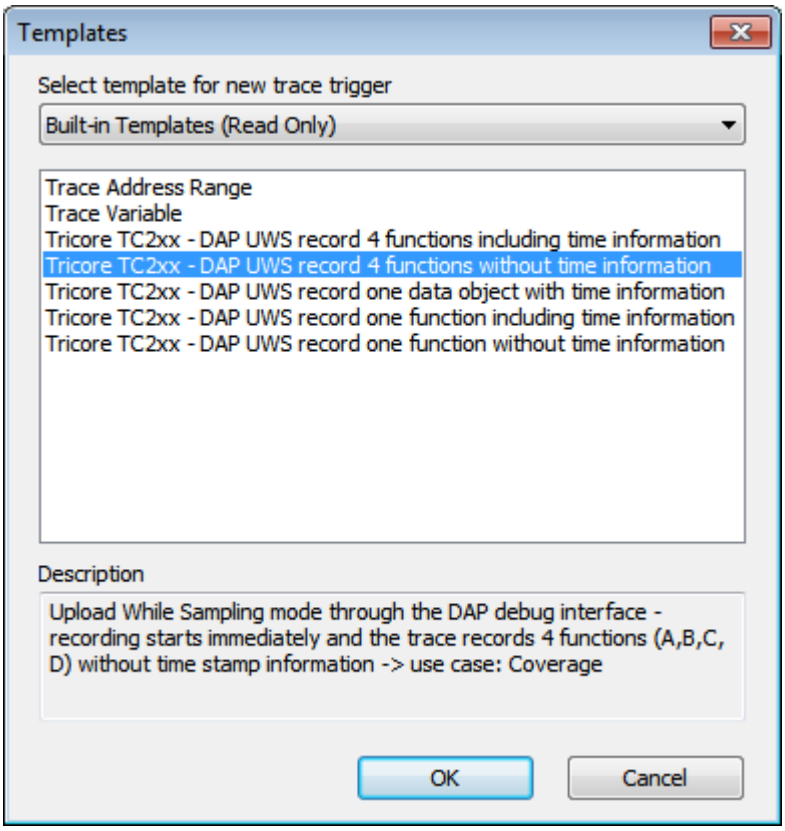


‘New Configuration’ dialog is opened where user can choose several Hardware Trigger Configuration possibilities. ‘Automatic’ and ‘Program flow + Instrumentation’ provide a preconfigured settings. ‘Wizard’ is also a very good starting point since it covers already few typical use cases.

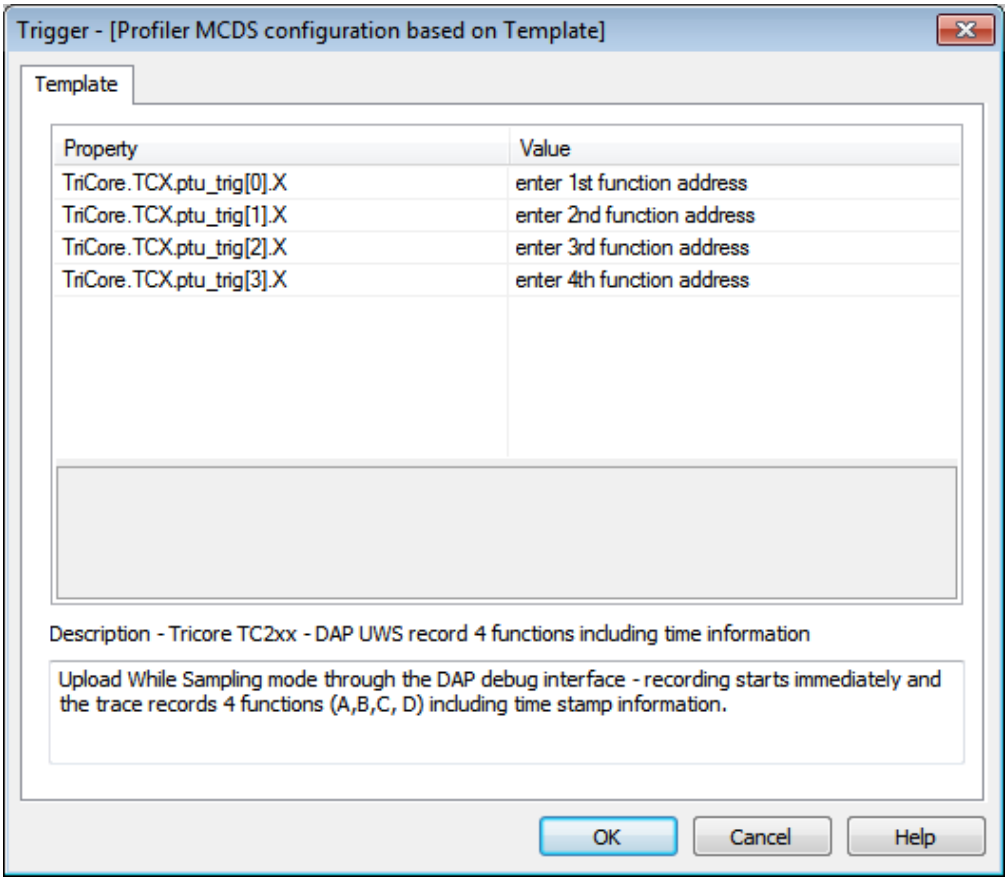
When someone would like to get the most out of the on-chip trigger/qualifier resources, the ‘Manual’ option should be selected. In this particular case, the ‘Template’ selection is selected since we will use one of the pre-built templates.



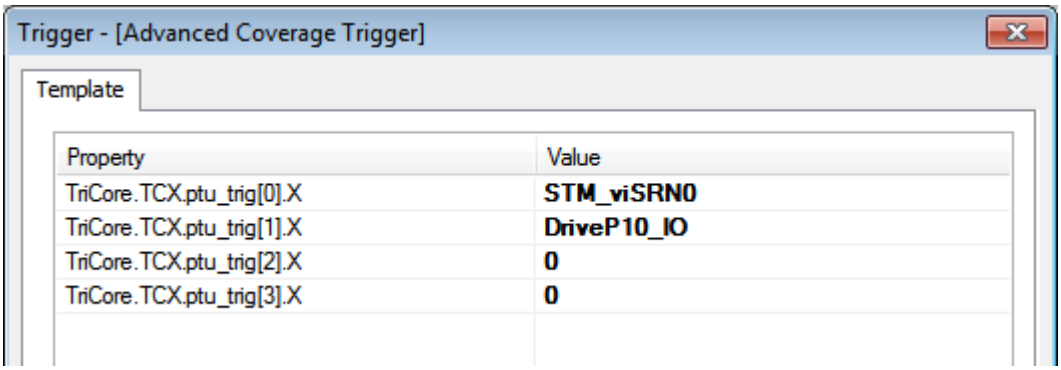
Next, select ‘Built-in Templates’ in the Templates dialog and pick ‘Tricore TC2xx – DAP UWS record 4 functions without time information’ pre-built template.


Templates
Select template for new trace trigger
Built-in Templates (Read Only)
Trace Address Range
Trace Variable
Tricore TC2xx - DAP UWS record 4 functions including time information
Tricore TC2xx - DAP UWS record 4 functions without time information
Tricore TC2xx - DAP UWS record one data object with time information
Tricore TC2xx - DAP UWS record one function including time information
Tricore TC2xx - DAP UWS record one function without time information
Description
Upload While Sampling mode through the DAP debug interface - recording starts immediately and the trace records 4 functions (A,B,C, D) without time stamp information -> use case: Coverage
OK
Cancel

After confirming the selection, a dialog opens where addresses for 4 functions are entered. If less than four functions need to be profiled, simply enter an address which is never executed (e.g. 0) for the redundant function/code addresses.


Trigger - [Profiler MCDS configuration based on Template]
Template
Property
Value
TriCore.TCX.ptu_trig[0].X
enter 1st function address
TriCore.TCX.ptu_trig[1].X
enter 2nd function address
TriCore.TCX.ptu_trig[2].X
enter 3rd function address
TriCore.TCX.ptu_trig[3].X
enter 4th function address
Description - Tricore TC2xx - DAP UWS record 4 functions including time information
Upload While Sampling mode through the DAP debug interface - recording starts immediately and the trace records 4 functions (A,B,C, D) including time stamp information.
OK
Cancel
Help

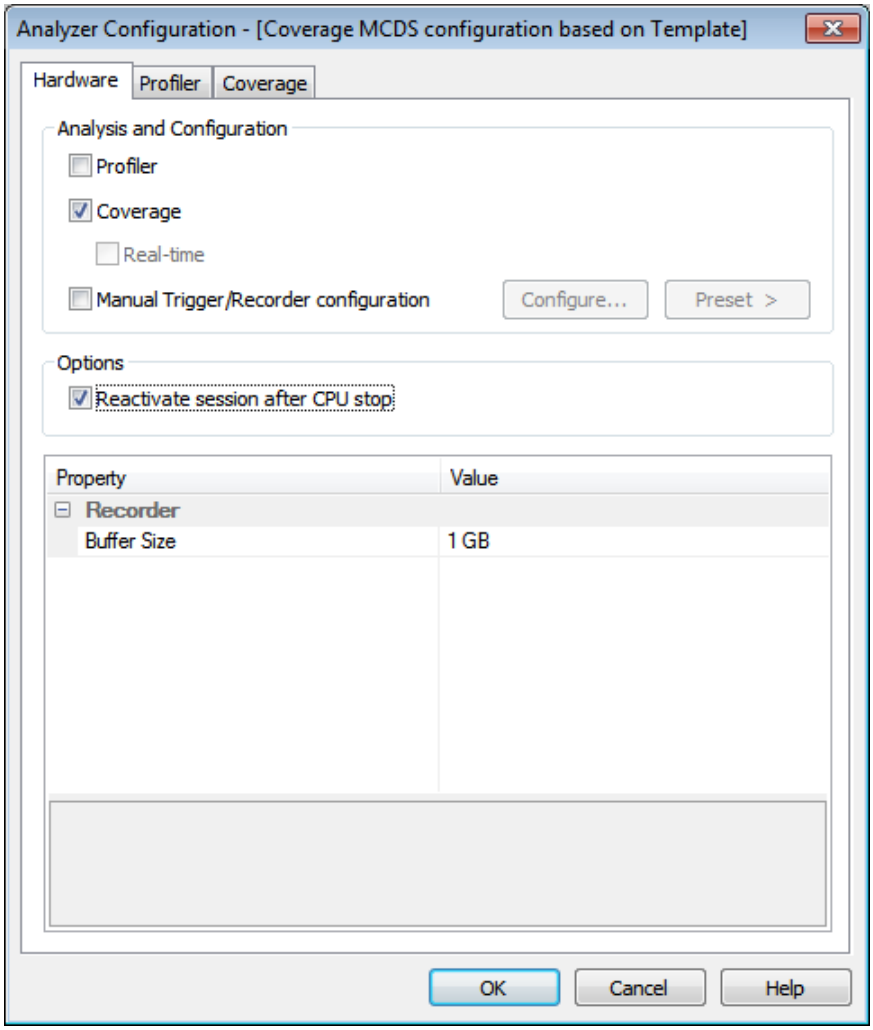
Next picture shows the configuration for two functions.



Close the configuration and run the coverage.

Note: Less functions are tested and less often they are called, the longer the coverage session will be.

When complete application must be tested for coverage, uncheck the 'Manual Trigger/Recorder configuration' and winIDEA will configure on-chip trace accordingly.



12 Getting Started

1) Connect the system
2) Make sure that the target debug connector pinout matches with the one requested by a debug tool. If it doesn't, make some adaptation to comply with the standard connector otherwise the target or the debug tool may be damaged.
3) Power up the emulator and then power up the target.
4) Execute debug reset
5) The CPU should stop on reset location.
6) Open memory window at internal CPU RAM location(s) and check whether you are able to modify its content.
7) If you passed all 6 steps successfully, the debugger is operational. Now you may add the download file and load the code to the RAM
8) To program the flash or download the code to the RAM, which is not accessible after reset, make sure you use the initialization sequence to enable the access. First, the debugger executes reset, then the initialization sequence and finally the download or flash programming is carried out.

13 Troubleshooting

- When the application behaves unpredictably when the debugger is connected, close all debug windows and try again to run the application from the reset state on.
- Try 'Slow' JTAG Scan speed if the debugger cannot connect to the CPU.
- Make sure that the power supply is applied to the target JTAG connector when 'Vref' is selected for Debug I/O levels in the Hardware/Emulator Options/Hardware tab, otherwise emulation fails or may behave unpredictably.
- When flash programming fails, double check that proper target device is selected in winIDEA.
- When performing any kind of checksum, remove all software breakpoints since they may impact the checksum result.
- Trace, Profiler and Coverage functionality are available only on (ED) Emulation Devices.
- Full Emulation Memory can be also used for other purposes but tracing such as calibration, code, constants or data storage. When full Emulation Memory is used for other purposes, trace and functionalities based on trace such as profiler and coverage cannot be used. Available Emulation Memory tiles can be split between different use purposes. Amount of Emulation Memory, which target application doesn't use, respectively makes available for tracing, must be specified in the Hardware/Emulation Options/CPU Setup/Advanced Options (see chapter 3.4 for more details). More Emulation Memory is allocated for the trace buffer, the longer trace, profiler and coverage session will be. Be cautious when Emulation Memory is being used by the application. If the target application would inadvertently use the same tiles of Emulation Memory, which were allocated for tracing, trace could behave unexpectedly or would not work at all. Note that all this is not applicable when tracing through the Aurora trace port.

Disclaimer: iSYSTEM assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information herein.

© iSYSTEM. All rights reserved.